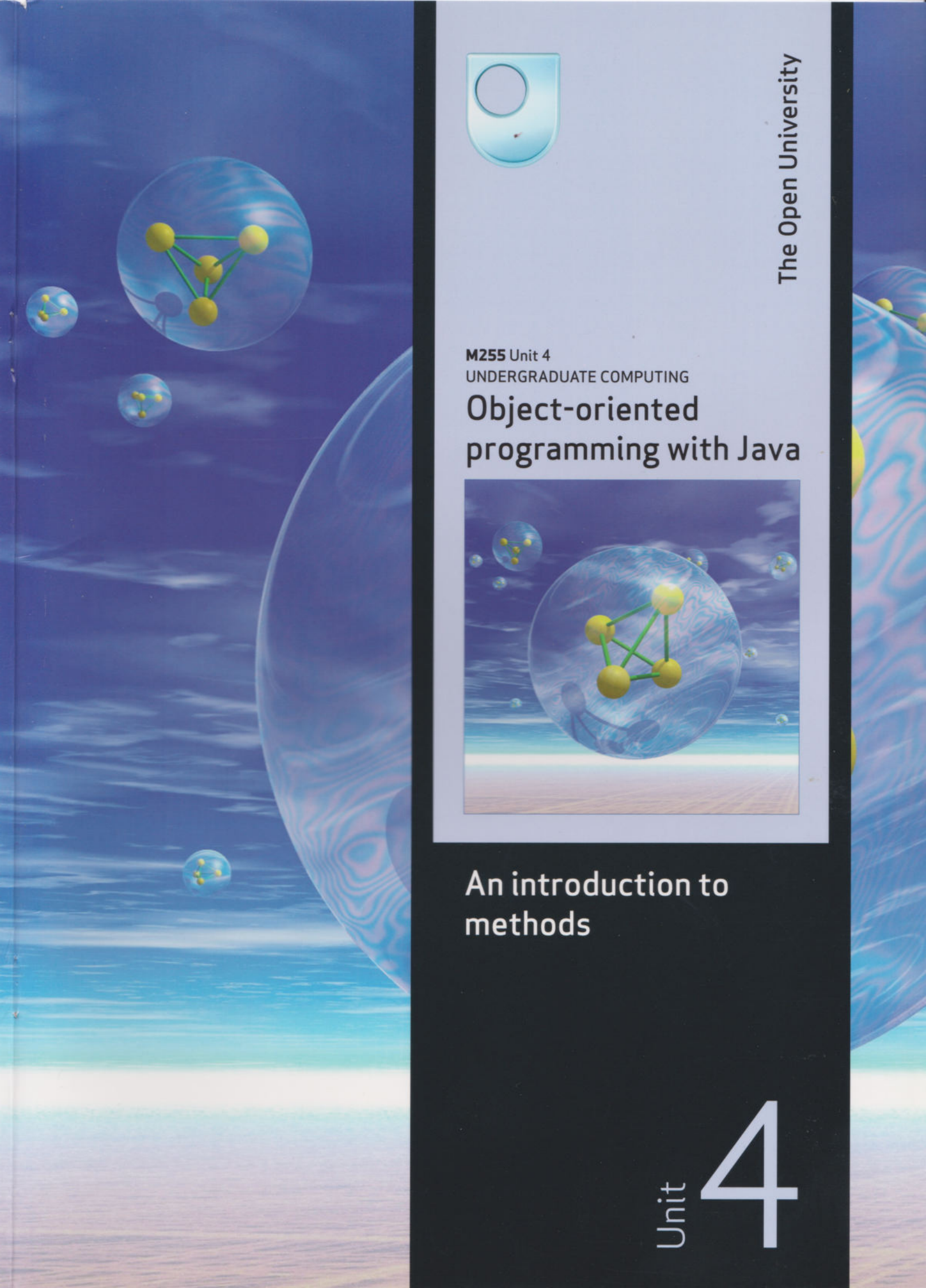


The Open University

M255 Unit 4
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

An introduction to methods

Unit 4

M255 Unit 4
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

An introduction to methods

Unit 4

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5496 4

2.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

In earlier units you interacted with objects by sending them messages. In this unit you are going to use the OUWorkspace as before, but you are also going to be looking 'behind the scenes' at the code that causes `Frog` objects to behave as they do; in other words, you will see how objects are programmed to respond to messages. To do this you will use an **editor** to view and to modify the source code for the `Frog` class. As you know from *Unit 1*, your computer does not understand the source code in the form you have written it so, when you have modified the code, you will need to **compile** it using the Java **compiler** in BlueJ.

The behaviour of an object is determined by its protocol – the messages to which it can respond. In this unit you will learn how to extend the protocol of instances of a particular class by providing the class with the code that an object of that class needs to execute in response to these new messages. To add a new message to the protocol of objects of some class we must write a piece of program code called a **method**. This is the code that is executed when the corresponding message is sent to an instance of that class. Thus creating new behaviours is about writing new methods or changing existing ones.

In *Unit 2* you met polymorphism, an important concept in object-oriented programming. This unit will introduce you to two further principles: **encapsulation** and **reuse of code**.

In the remainder of this unit we aim to:

- introduce methods and the editor in BlueJ (Section 1);
- introduce instance variables and their accessor messages (Section 2);
- explore more complex methods, including those with return values and arguments (Sections 3 and 4);
- introduce the principles of reuse of code and encapsulation (Sections 5 and 6).

Sections 1 to 4 consist of text (including SAQs), activities (for which you will need your computer) and paper-based exercises. These sections may look short on paper, but the activities will take time to complete as you will be building up facility with the BlueJ editor and the techniques necessary for writing code. Sections 5 and 6 are mainly discursive, with few exercises or activities, and so may not take you as long to complete. However, the concepts they introduce are fundamental to your understanding of the principles of object-oriented programming. Section 7 is a consolidation section. It consists of a few, longer, activities and aims to draw together in one extended example all the techniques that you have learned in this unit. It is important to complete this section as it will give you practice in the type of activities that may arise in practical TMA questions.

1 Classes and methods

In this section we shall start by exploring the code for the `Frog` class. We then extend the protocol for `Frog` objects by adding a new method, `catchFly()`, and add a new method, `doubleLeft()`, to the `Toad` class. Two discussions complete the section: a short look at documenting a class, and a description of the process that starts with editing the code and results in observable behaviour.

1.1 A look inside the `Frog` class

First, we will look at how the `Frog` class is constructed. We will use the editor in BlueJ to scroll through the source code for the `Frog` class and look at the various parts. In this unit we will only be concerned with changing the code for some methods and writing some new methods, so the explanations for the various parts of the code are only sufficient to enable you to carry out these tasks.

ACTIVITY 1

This activity introduces you to the way classes are written by way of an exploration. As such, no specific 'exercise' is set, and the activity contains its own discussion of what you see via the exploration.

Launch BlueJ and open the project named Unit4_Project_1 from the Block1 folder. You should see a window containing rectangles representing the classes `Toad`, `Frog` and `HoverFrog`.

If you cannot remember how to do this, refer to the *Software Guide*.

Double-click on the `Frog` class to open the editor. The window that opens should look something like Figure 1. If it does not, make sure the text in the drop-down list at the top right-hand corner of the window contains the word 'Implementation'.

Your editor window should now look like that in Figure 1.

Figure 1 shows the major elements of the code for the class `Frog` and a brief explanation for each part. In this section we shall be concentrating on the methods. For now, just make sure you can identify the different parts of the code in the editor window on your screen, and then close the editor and return to reading this text.

This representation uses a notation called UML (Unified Modelling Language), which is commonly used to diagrammatically model the relationships between classes. You will notice that the classes `Frog` and `HoverFrog` are connected by an open-headed arrow. This indicates that `HoverFrog` is a subclass of `Frog`.

The Java keyword `import` tells the Java compiler that it needs to find some libraries which contain existing code that it will need to use. We shall discuss the use of Java libraries later.

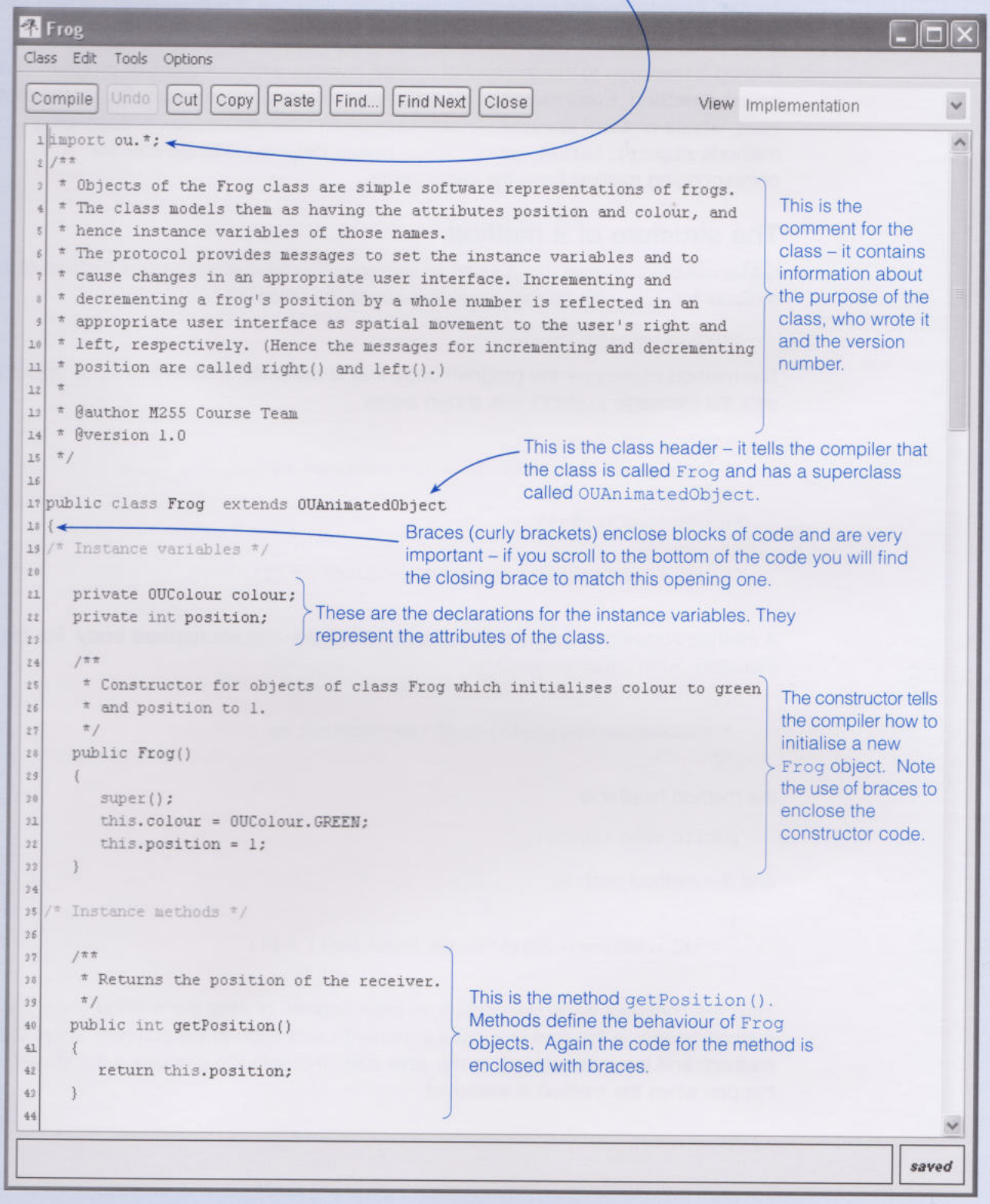


Figure 1 An editor window for the `Frog` class

1.2 Messages and methods

In the activities in the previous units you sent messages to `Frog` and `Toad` objects that caused them to behave in a certain manner; for instance, if a `Frog` object is sent the message `jump()` it will seem to jump in the Amphibians graphics window.

Adding a message to the protocol of a class involves writing a piece of program code called a **method**. Every message must have a corresponding method. So, for the class `Frog`, whose protocol is `right()`, `left()`, `green()`, ... , there are corresponding methods `right()`, `left()`, `green()`, ... Notice that the message and the corresponding method have the same name.

The structure of a method

Let us use the message `right()` as an example. You have already used this part of the protocol for `Frog` with message-sends such as the following.

```
frog1.right();
```

The method `right()` – the program code that is executed whenever a `Frog` object is sent the message `right()` – is shown below.

```
/**
 * Increments the position of the receiver by 1.
 */
public void right()
{
    this.setPosition(this.getPosition() + 1);
}
```

A method consists of a **comment**, a **method header** and the **method body**. For the method `right()` the comment is:

```
/**
 * Increments the position of the receiver by 1.
 */
```

the method header is:

```
public void right()
```

and the method body is:

```
{
    this.setPosition(this.getPosition() + 1);
}
```

Thus the comment is the description, in plain English, of what the method does; the method header is the part that gives important information to the compiler about the method; and the method body is the Java code that tells the compiler what should happen when the method is executed.

SAQ 1

The following is the method `home()` in the class `Frog`.

```
/**
 * Resets the receiver to its "home" position of 1.
 */
public void home()
{
    this.setPosition(1);
}
```

Write down:

(a) the comment;

(b) the method header;

(c) the method body.

ANSWER

(a) The comment is:

```
/**
 * Resets the receiver to its "home" position of 1.
 */
```

(b) The method header is:

```
public void home()
```

(c) The method body is:

```
{
    this.setPosition(1);
}
```

The method comment is enclosed by `/**` and `*/`. This is so that the documentation for the method can be created automatically, as you will see later. Within the method body, single-line comments are sometimes used to explain what is intended by a section of code. They start with // and do not appear in the automatically generated documentation.

The method header has three parts. The first part is the *access modifier*; it tells the Java compiler what other objects can send the corresponding message. For now, all the methods we use will be `public` – any object may send the corresponding message to objects of the class defining the method (including you, from the OUWorkspace!). Declaring a method as public also enables an object of any subclass of the class defining the method to respond to the corresponding message. The second word tells the Java compiler whether the method returns a value as a message answer and, if so, the type of the value returned. The methods we consider in this section return no values – this is indicated by the keyword `void`. For example, the method `right()` returns no value, so the header includes the word `void`. The final word is the name of the method, in this case `right`, followed by parentheses. When you look at some more complicated methods there will be arguments between the parentheses, but for now all you need to remember is that you must put the parentheses after the name of the method.

The method body consists of one or more Java statements, each ending with a semicolon. These statements comprise the code that is executed whenever the corresponding message is sent to a `Frog` or `HoverFrog` object. We say that when a message is sent to an object at run-time, the corresponding method of the same name is executed (or 'invoked'). The method body is enclosed by braces (curly brackets).

Remember that a Java statement is a single instruction for the Java compiler.

Remember from *Unit 3* that a variable is a named memory location used to store a value, or reference to an object, which can be changed by assignment. The variable `this` is known as a pseudo-variable: it can only reference a particular object – the receiver – and you cannot change (by assignment) the object to which `this` refers.

The special variable `this` is used within a method whenever there is a need to reference the object that was sent the corresponding message and so caused the method to execute. In other words, `this` references the receiver of the message that caused the execution of the method. You do not (indeed cannot) declare the special variable `this`, but it is always automatically available for use with a method and is an example of a **pseudo-variable**.

Each message in the protocol of `Frog` has a corresponding method that is similar in its overall structure to that of the method `right()`. A method can be thought of as the way the behaviour in response to a message is provided – the concrete implementation of a message.

SAQ 2

What is the difference between a message and a method?

ANSWER

A method is the piece of code (in the case of M255, Java code) that is executed when an object receives a particular message. In other words, when you send a message to an object, it is the corresponding method code that is executed at run-time.

Exercise 1

The comment and method header for the method `left()` of the class `Frog` is given below.

```
/**
 * Decrements the position of the receiver by 1.
 */
public void left()
```

The code for the method `left()` consists of a single Java statement (similar to that for `right()`). Write down the code. Do not forget the braces and the semicolon.

Solution

```
{
    this.setPosition(this.getPosition() - 1);
}
```

ACTIVITY 2

This activity introduces you, through a second exploration using the BlueJ editor, to the way methods are written. Again, no specific 'exercise' is set and the activity contains its own discussion of what you see via the exploration.

Launch BlueJ (if you do not already have it running) and open the project named Unit4_Project_1 from the Block1 folder. Then double-click on the class `Frog` to open the editor. From now on we will just ask you to open the BlueJ editor on a class in a particular project. You may need to make a note of how to do this.

Open the BlueJ editor on the class `Frog` in Unit4_Project_1. The instance methods are all set out after the constructor. Scroll down to the method `left()`. You will see the following code.

Find the comment (the text between /** and */), the method header and the method body (the part enclosed in braces).

```
/**
 * Decrements the position of the receiver by 1.
 */
public void left()
{
    this.setPosition(this.getPosition() - 1);
}
```

Browse through all the methods that are available for `Frog` objects. Note how all the methods have the same format: the method header followed by the method body. Do not worry if you cannot understand the code at the moment – all will be revealed in the next few units.

Now open an editor window on the class `Toad` and see how the method `left()` for `Toad` differs from the method `left()` for `Frog`. Try to arrange matters so that you can see copies of the methods together. (You can open an editor window for each class and have them on the screen at the same time.)

Coding a new method

Let us now consider a message that is not yet part of the protocol for the `Frog` class. Suppose that we want `Frog` objects to respond to a new message called `catchFly()`. The effect of `catchFly()` – the new behaviour it will provide – is that a `Frog` object which receives a `catchFly()` message will perform a jump, perform a croak, and then move one position to the right. In order to extend the protocol of `Frog` objects in this fashion, a method for `catchFly()` must be provided.

Exercise 2

Write down a comment and method header for the method `catchFly()`.

Solution

Here is our solution; your comment will probably differ but should provide the same information.

```
/**
 * Performs a jump, then croaks and then moves right.
 */
public void catchFly()
```

Exercise 3

Write down the Java statements that will complete the method `catchFly()`. Remember the braces and the semicolons.

Solution

```
{
    this.jump();
    this.croak();
    this.right();
}
```

You are now going to put these ideas about messages and methods into practice in the following activities.

ACTIVITY 3

You are now going to create a new method, `catchFly()`, for the `Frog` class. Here is the complete code for `catchFly()`, as developed above:

```
/*
 * Performs a jump, then croaks and then moves right.
 */
public void catchFly()
{
    this.jump();
    this.croak();
    this.right();
}
```

In order to create a new method, you need to edit the source code for the class. Open the editor on the `Frog` class in Unit4_Project_1. Scroll the editor window down to the end of the code. What you see now should look like Figure 2.

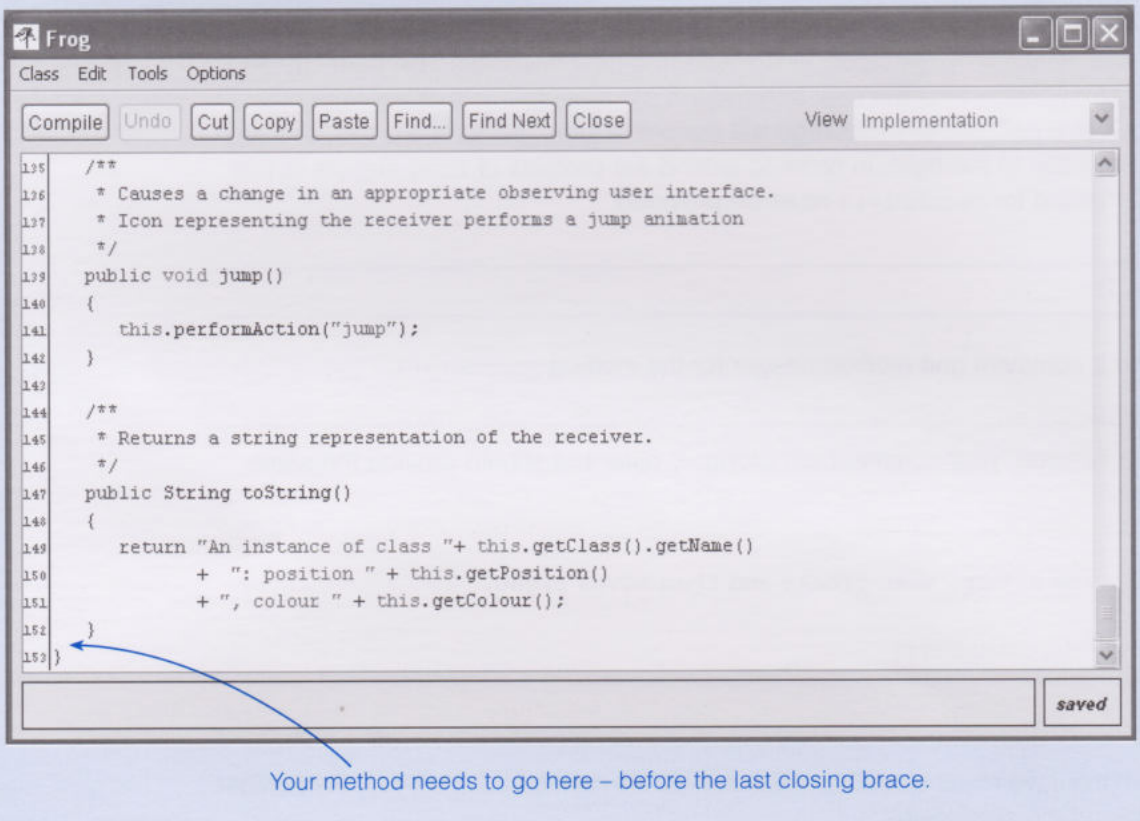


Figure 2 The **Frog** class before the method **catchFly()** has been inserted

Position your cursor after the brace that ends the last method in the class. There will be a brace below your cursor. This is correct – the class needs a final closing brace. Insert the method code for `catchFly()`. Your editor window should now look like Figure 3.

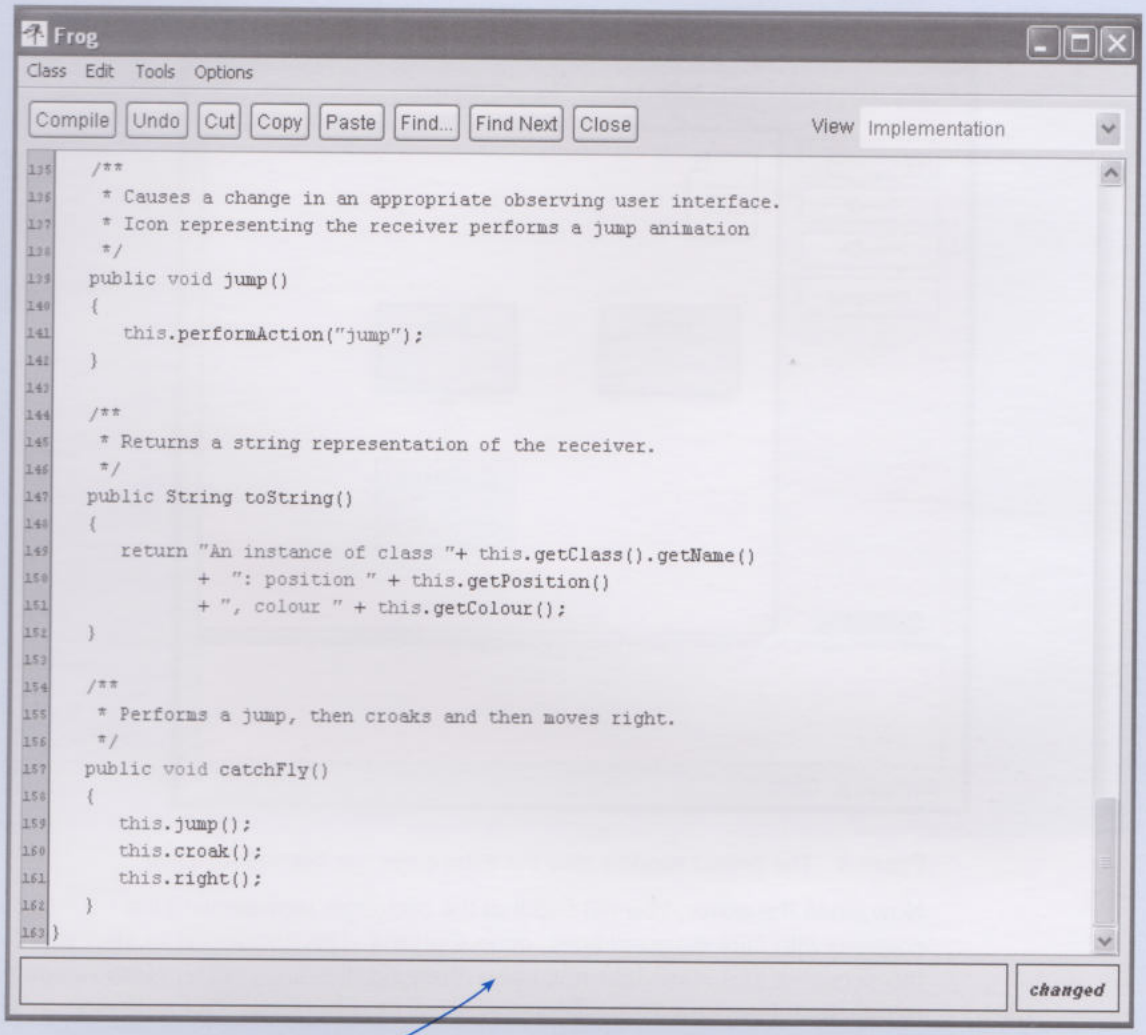


Figure 3 The `Frog` class after the method `catchFly()` has been inserted

Now click on the Compile button at the top-left of the editor window, and look at the message area at the foot of the window. You should see either the word 'compiling ...' or the phrase 'File saved' in the message area. After a short time that will be replaced by 'Class compiled – no syntax errors'. If you have made a mistake when typing in the method `catchFly()`, an error message will appear in the message area and the code somewhere near the error will be highlighted. Check your code very carefully, making sure you have placed the semicolons and braces correctly and used correct capitalisation, then press the Compile button again. This process of finding errors, correcting them and recompiling until the code compiles without errors is known as **debugging**. Continue debugging until you have no more errors.

In fact you could have placed our new method after any of the existing methods in the class. The order in which the methods appear in a class does not matter. Usually methods are grouped together in what seems a logical order to the person writing the code. As `catchFly()` does not seem to belong to a logical grouping with any existing methods, we have asked you to place it at the end of the class.

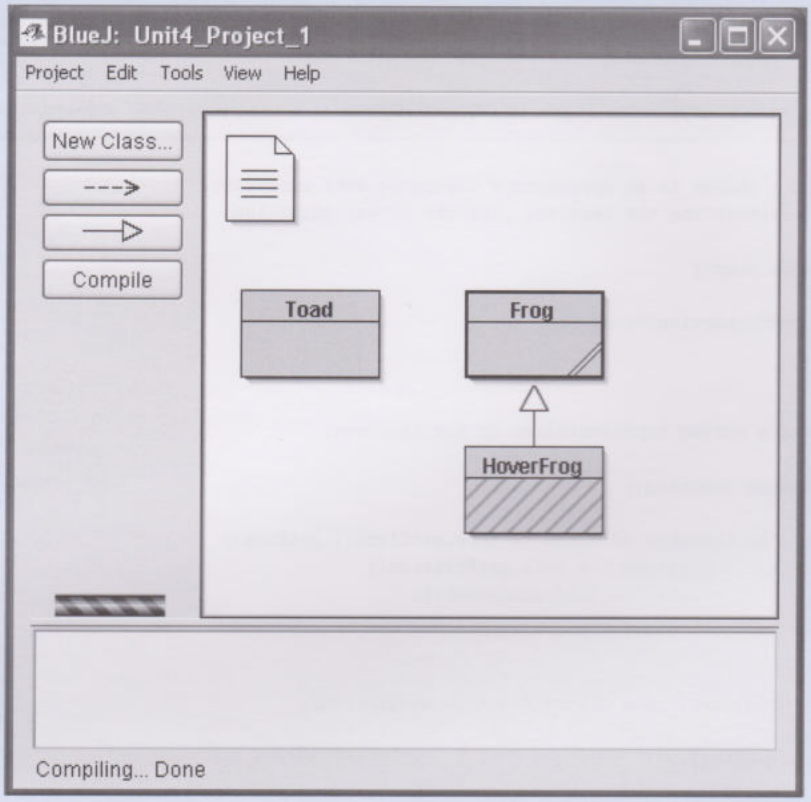


Figure 4 The project window after the `Frog` class has been recompiled

Now close the editor. You will find that the rectangle representing the `HoverFrog` class is covered with blue diagonal lines, as in Figure 4. This shading is to alert you that, as it is the subclass of a class that has been changed, the `HoverFrog` class needs to be recompiled. Press the Compile button in the BlueJ window. The `HoverFrog` class should now look like the other classes.

Congratulations – you have created your first method in the BlueJ Java environment.

ACTIVITY 4

You now need to test your new method in order to ensure that it produces the desired behaviour when the message `catchFly()` is sent to a `Frog` instance. To do that, you will need to create some `Frog` objects in the OUWorkspace.

In future we will just ask you to execute statements in the OUWorkspace and to view their effect in the graphical display. You may need to make a note of how to do this.

With Unit4_Project_1 open, select OUWorkspace from the BlueJ Tools menu. An OUWorkspace window will appear. In order to see, graphically, the effect of message-sends to `Frog`, `Toad` and `HoverFrog` objects, you will need to open an Amphibians window. Do this by selecting Open from the Graphical Display menu in the OUWorkspace.

To execute code, select it and then either press Ctrl+E or select Execute Selected from the Action menu.

In the Code Pane of the OUWorkspace window, type the following code and execute it.

```
Frog kermit = new Frog();
```

You should see the variable name `kermit` appear in the Variables Pane, and the frog it references should appear in the Amphibians window.

Create some more `Frog` objects in the same way. Send each object a `catchFly()` message using message-sends such as

```
kermit.catchFly();
```

and observe their behaviour.

Now do the same for `HoverFrog`. Create some `HoverFrog` objects and send them `catchFly()` messages. Observe the result.

Finally, create a `Toad` object called `croaker`. Execute the following statement in the workspace and observe what happens.

```
croaker.catchFly();
```

Try to explain why there is a difference between what happens when you send a `catchFly()` message to a `Frog` or `HoverFrog` object and when you try to send the same message to a `Toad` object.

DISCUSSION OF ACTIVITY 4

You may have noticed several things.

All the instances of the class `Frog` that you create can respond to a `catchFly()` message.

`HoverFrog` instances respond to the message `catchFly()`. This is a consequence of the class/subclass relationship. `HoverFrog` is a subclass of `Frog` and whatever behaviour is defined for `Frog` is also available for `HoverFrog`.

The statement `croaker.catchFly();` does not execute correctly. Instead you get the following error message in the Display Pane.

```
Semantic error: Message catchFly() not understood by class 'Toad'.
```

Note that if you select more than one line of code to execute in the OUWorkspace, the error message displayed includes the line number on which the error occurred. For example, if you were to select and execute the code

```
Toad croaker = new Toad();
croaker.catchFly();
```

all in one go, then you would get the error message

```
Semantic error: line 2. Message catchFly() not understood by class 'Toad'.
```

The error message occurs because `Toad` is not a subclass of `Frog` and therefore the behaviour defined for `Frog` objects will not be automatically defined for `Toad` objects.

ACTIVITY 5

This activity continues to use the project Unit4_Project_1.

You are now going to write a new method, called `doubleLeft()`, for the class `Toad`. This method will decrement the value of the instance variable `position` by 4. However, before defining this new method, satisfy yourself that the `Toad` objects do not currently respond to a `doubleLeft()` message.

1 In the OUWorkspace, execute the following statements.

```
Toad toad1 = new Toad();
toad1.doubleLeft();
```

2 We will now guide you through the code for the new method.

The comment and method header for your new method `doubleLeft()` are given below.

```
/**
 * Moves the receiver left twice.
 */
public void doubleLeft()
```

Open the editor on the `Toad` class. Start writing the new method at the end of the methods in the `Toad` class, just as you added the method `catchFly()` to the `Frog` class. You now need to write the method body to achieve the effect indicated in the comment. This can be done by sending the message `left()` twice to the object referenced by `this` (the object whose method is being executed).

Complete your method by writing the two statements and the enclosing braces. Compile the `Toad` class (debugging any errors you may have made).

3 Test your new method by sending `doubleLeft()` messages to some `Toad` instances you create in the OUWorkspace. You might like to use the following messages to start with.

```
Toad toad1 = new Toad();
toad1.doubleLeft();
```

DISCUSSION OF ACTIVITY 5

1 You should obtain the following error message.

```
Semantic error: Message doubleLeft() not understood by class 'Toad'.
```

2 Here is the complete method `doubleLeft()`.

```
/**
 * Moves the receiver left twice.
 */
public void doubleLeft()
{
    this.left();
    this.left();
}
```

3 Check that your message-sends have worked by inspecting the instance variable `position` for `toad1` or by watching the progress of the toad icon in the Amphibians window.

Now you have completed these activities, you have:

- used the BlueJ editor and examined existing methods for a selected class;
- defined new methods for a class, compiled and tested them;
- seen that, if a method is defined for a particular class, and declared as `public`, then the corresponding message can be sent to objects of that class and its subclasses, but not to objects of other classes.

If you have had difficulties with Activities 3–5, Unit4_Project_2 incorporates all the changes made to the classes `Frog` *and* `Toad` *in this set of activities.*

SAQ 3

What do you need to do to add a new message to the protocol of a class?

ANSWER.....

Adding a message to a protocol requires writing a new method for the class. The editor is used to do this.

SAQ 4

How can an object send a message to itself?

ANSWER.....

By using the special reference variable `this` as the receiver of the message.

1.3 Documenting methods

Each time we have asked you to write a method, we have given a description of what the method is intended to do in the form of a comment. This comment is given in a particular format so that it can be used by a program called **Javadoc**, which comes with Java. The Javadoc program picks up information from specially formatted comments and other parts of the class code, such as the constructor and the method headers. These are all used to create an HTML file, which describes the class in a standard way. This description is aimed not at the Java compiler, but at human readers (and possibly the writer of the code at a later date, when he or she might well have forgotten what the methods do).

ACTIVITY 6

Open Unit4_Project_2. This project incorporates all the changes made to the classes `Frog` and `Toad` in Activities 3–5.

From the Tools menu select Project Documentation (or press Ctrl+J on your keyboard).

(A dialogue box may appear; if it does, click on Regenerate.)

This will launch your web browser, and after a short while a browser window will appear containing the Javadoc information for the project. The left-hand frame of the browser window will display an alphabetical list of all the classes in the project. The right-hand pane will display the documentation for the class, which is at the top of the list in the left-hand frame, in this case the documentation for the class `Frog`. Clicking on a class name in the left-hand list will display the documentation for that particular class in the right-hand frame.

For now, ignore the menu items in the blue shaded area at the top of the right-hand frame. What these items mean and do are explained in the *Software Guide*.

Focus your attention on the text underneath the first solid line in the right-hand frame. You do not need to understand everything you see, but here is a short explanation of the main parts of the documentation that you might find useful as you scroll downwards.

The first section under the title Class Frog shows where `Frog` comes in the class hierarchy. It is a subclass of `OUAnimatedObject` and has one subclass, called `HoverFrog`.

The second section shows the class comment.

Next, the constructor summary displays the first sentence of the constructor's comment.

Following the constructor summary you are shown a summary of the methods, in alphabetical order. This summary just shows the name of the method, what arguments it takes and its return type, and the first sentence in the method's comment.

The inherited methods are displayed next. Clicking on a name of one of these inherited methods will display the documentation for the class that implements that method in your web browser.

Next, more detail of the constructor is displayed – this time you are shown the access modifier and the entire comment.

Finally, you are shown more detail about the methods defined in the class; you are shown the entire comment and the entire method header.

Look at the method `catchFly()`. You should see the comment you typed in as the description of the method.

Close the browser window, open the editor on the `Frog` class, and move the comment for `catchFly()` to after the method header. Recompile the `Frog` class. Did `Frog` recompile successfully?

Once you have got the class to recompile, select Project Documentation from the BlueJ Tools menu. This time a dialogue box will appear. You need to click on Regenerate. Look at the Javadoc entry for `catchFly()`. What do you notice?

Return the comment for `catchFly()` to the correct position and recompile the `Frog` class. Close the editor and the browser, and exit from BlueJ.

DISCUSSION OF ACTIVITY 6

You should have been able to recompile the `Frog` class with no difficulty – Java takes no notice of the comment or where it is placed. However, you should find that Javadoc has now failed to pick up the comment, and so it is missing from the entry for the method `catchFly()`.

Information will only be picked up by Javadoc if it is in a particular form and position. So far you have learned that, in order to ensure that the Java compiler takes no account of the method comment, but that Javadoc *can* pick up the comment:

- you must enclose the method comment between `/**` and `*/`; and
- you must place the method comment directly before the method header.

Method comments serve several purposes.

- They help focus your mind on the purpose of the method, i.e. on the behaviour the corresponding message will cause.
- Months later, when as a student you are revising, or as a programmer you are reviewing this method among dozens of other methods you have written previously, comments can be enormously helpful in giving you a quick reminder of what the code is about.
- They support reuse of code as they help anyone else looking at your code to understand how to use the method and what the method should do.

Notice that the method comment should not focus on *how* the code works; it should focus on the code's purpose or effect, on what is intended. You are best advised to think of the meaning of the message that the method implements, rather than how the method works. Although you should always include a method comment, you need not be so pedantic about including comments in other parts of the method. As a general rule, comments other than the initial one are more about how a method is implemented. You should use these when you think that something is tricky to understand, so that those who maintain the code will know what was intended.

1.4 Editing, compiling and executing methods

So far in this unit we have glossed over what happens when you write a piece of code for a method, compile it and then have it execute. The following is a reminder of the process that was described in *Unit 1*.

Java is a high-level programming language, i.e. it is designed so that human beings can read it. A Java program is written, or modified, using an editor. This could be something as simple as Notepad, but in this course you use the editor provided by the BlueJ environment. The code produced is called the source code. Earlier in this unit you used the editor to modify the source code for the `Frog` class by adding a method `catchFly()`.

The computer cannot run Java code directly. Before a method can be executed, the source code for the class it belongs to must be translated into machine code – the low-level language made up of the instructions that the computer understands at the most basic level. Compilers for many high-level languages go straight from source code to machine code, but this technique has the disadvantage that the resulting machine-code program will only run on one particular type of computer; computers with a different set of machine-code instructions will not run it.

The Java compiler takes a different approach. It translates the source code into an intermediate language called **bytecode**. In BlueJ, the compilation is done when the Compile button is pressed. In the case of the `Frog` class, this will create a bytecode file, `Frog.class`, from the source code file `Frog.java`.

The bytecode file is portable, because each computer that can run Java programs has a Java Virtual Machine (JVM) – which is itself a program – that understands bytecode and converts it into the machine code required for that particular computer. So any computer that runs Java will be able to execute our compiled class `Frog`.

At run-time, when an object receives a message, the Java Virtual Machine selects the appropriate method to be executed (determined by the class of the object). We call this *invoking* the method or **method invocation**. The first time a method is used at run-time, the JVM converts the corresponding bytecode into the machine code appropriate for that particular computer.

Once the bytecode for a method has been translated into machine code, the JVM retains the machine code for the next time the method is required, so it does not have to do the translation over and over again. The machine code is only discarded when you exit from BlueJ, or if you modify and recompile the class (when, obviously, the low-level instructions will need to be regenerated). If a method is never used at run-time, it is never compiled to machine code.

Java programmers almost never have to be concerned with bytecode – the compiler and the Java Virtual Machine deal with all that side of things. However, purely for interest, here is a snippet of bytecode. Can you guess what it is?

```
 0: aload_0
 1: aload_0
 2: invokevirtual #28; //Method getPosition:()I
 5: iconst_1
 6: isub
 7: invokevirtual #27; //Method setPosition:(I)V
10: return
```

2 Instance variables

You have seen that, when a message is sent to a `Frog` object, a method with the same name is executed. Some of these methods, such as `left()`, result in changing the state of the object. The current state of an object is represented by the values of its **instance variables**.

You have already seen a pictorial representation of instance variables in *Unit 3*. Here is a diagram that represents the initial state of a `Frog` object.

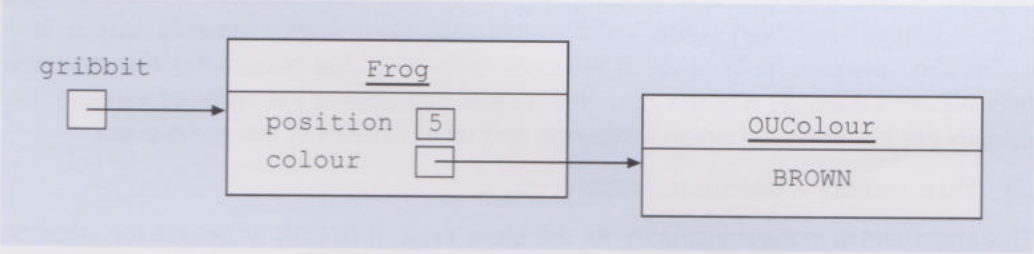


Figure 5 Diagrammatic representation of a `Frog` object

This section looks at the role of instance variables and the use of accessor methods for instance variables. It concludes by showing how to add a new instance variable to a class.

2.1 Instance variables and methods

As you will see in Activity 7 and the corresponding discussion, the definition of a class begins by declaring its instance variables.

Each instance variable, after initialisation by the constructor, either contains a reference to an object or a value of some primitive data type. For instance, after a `Frog` object is initialised, the instance variable `colour` contains a reference to an instance of the class `OUColour`, in this case `OUColour.GREEN`; the instance variable `position` contains a value of the primitive type `int`, in this case `1`.

Note carefully that, in contrast with an accessor message, the written notation for directly accessing an instance variable does not include parentheses. So within a method, the expression `this.colour` is immediately recognisable as a direct access of the `colour` instance variable of the object referenced by `this`. On the other hand, `this.getColour()` is a message-send, which answers with the value of the receiver's `colour` instance variable.

ACTIVITY 7

Open the editor on the class `Frog` in the project Unit4_Project_2. Find all the Java statements that use the instance variable `colour`. What do you think each statement does?

DISCUSSION OF ACTIVITY 7

The Java statements that use the instance variable `colour` are as follows.

(i) `private OUColour colour;`

This appears immediately after the class header and before the constructor. These statements are the variable declarations that define the instance variables for all `Frog` objects. That is, they tell the Java compiler the name of each instance variable and the type of values it can hold or reference. In this case, the instance variable `colour` is defined to reference an instance of the class `OUColour`. The access modifier `private` tells the Java compiler that the only objects that can access the instance variable directly are the object to which it belongs and other objects of the same class.

(ii) `this.colour = OUColour.GREEN;`

This statement is in the constructor for the class `Frog`. It directly accesses the instance variable `colour` and assigns it the object `OUColour.GREEN`.

(iii) `return this.colour;`

This statement is in the method `getColour()`. It directly accesses the `colour` instance variable of the receiver and returns it as the message answer. We discuss methods that return values in Section 3.

(iv) `this.colour = aColour;`

This statement is in the method `setColour()`. It directly accesses the instance variable `colour` and assigns it the `OUColour` object referenced by `aColour`, which is the argument of the message that caused the method to be executed. We discuss methods with arguments in Section 4.

Remember that although we talk about objects, we program them via their class.

A class defines the methods that can be executed for any instances of that class (or its subclasses). These methods are then stored in the compiled class, not in any instances of that class. However many instances of the `Frog` class there are, each method is stored once only – in the class. Hence when at run-time the JVM determines that an object is receiving a message, the JVM must determine the class of that object and then look in that object's class to find the corresponding method to execute. If it cannot find the method there, it will look in the superclass and so on. This description of how methods get to be executed is specific to Java; more generally and informally, we can (and will) say that when an object receives a message the object itself looks in its class (and, if necessary, its superclass and so on) to find the corresponding method to execute. (You will learn about executing methods from a superclass in *Unit 6*.)

Contrast this to how instance variables are stored. A class simply specifies what instance variables objects of that class should have. Each instance of that class then stores its own separate set of instance variables that conform to those specifications.

We now look at what happens when an object receives a message and the corresponding method body is executed.

The message-send `kermit.left()` causes the method `left()` to be executed, the code of which is shown below.

```
public void left()
{
    this.setPosition(this.getPosition() - 1);
}
```

Note the use of the **pseudo-variable** `this`. It is used like an extra argument to the method – it references the object that was sent the `left()` message (and hence caused the method to execute) so enabling the method to send the `setPosition()` and `getPosition()` messages to the correct object.

1 The `getPosition()` method is executed first. The code for the method is shown below.

```
public int getPosition()
{
    return this.position;
}
```

Notice again the use of `this` to reference the object that was sent the `getPosition()` message, so enabling the `getPosition()` method to access the instance variable `position` stored for that particular object. The value of the `position` instance variable is then returned as the message answer.

2 The next step is to take 1 away from the value that has just been returned by `getPosition()` and use this value as the argument to the message `setPosition()`.

3 The final step is that the method `setPosition()` executes. The code for the method is shown below.

```
public void setPosition(int newPosition)
{
    this.position = newPosition;
}
```

Again, `this` is used so that the instance variable `position` for the object that was sent the `setPosition()` message can be set to the value that was calculated in step 2.

Instance variables must be declared before they are used. You have seen the form of the declaration in Activity 7 and the corresponding discussion. Instance variables should also be given initial values, and we look at how this can be done in the next subsection.

2.2 Constructors and initialisation

When an instance of a class is created – using `new` – it has to be initialised. To initialise an object you need to initialise its instance variables. This is done by a constructor.

You learned about how the compiler uses the constructor in response to `new` in *Unit 3*, Subsection 1.4.

SAQ 5

Write down a Java expression to create a new instance of the class `Frog` referenced by a newly declared variable `aFrog`.

ANSWER

```
Frog aFrog = new Frog();
```

As an example of how to write a constructor, we look at the constructor for the `Frog` class in Exercise 4.

Exercise 4

The constructor is as follows.

```
/**
 * Constructor for objects of class Frog which initialises colour
 * to green and position to 1.
 */
public Frog()
{
    super();
    this.colour = OUColour.GREEN;
    this.position = 1;
}
```

The statement `super();` will be explained in *Unit 6*; for the moment we will ignore it. What is the purpose of the other two statements?

Solution

`this.colour = OUColour.GREEN;`

This statement sets the instance variable `colour` to the initial colour `OUColour.GREEN`.

`this.position = 1;`

This statement sets the instance variable `position` to the initial home position, `1`.

The effect of these two statements is to initialise the two instance variables, `colour` and `position`. Initialising instance variables is commonly done by the constructor.

2.3 Accessor methods

You are now going to explore further what you have just learned about instance variables by looking at the methods used to access them.

ACTIVITY 8

You have seen in Activity 7 that the method `setColour()` sets the instance variable `colour`. It does this by means of the following assignment statement.

```
this.colour = aColour;
```

Are there any other methods defined for the `Frog` class that also change the value of instance variables directly by assignment?

Use the BlueJ editor to examine all the methods defined for `Frog` in the project Unit4_Project_2.

DISCUSSION OF ACTIVITY 8

You should have identified `setPosition()` as the only other method that sets an instance variable using assignment. Here is the code for `setPosition()`.

```
/**
 * Sets the position of the receiver to the value of the argument
 * aPosition.
 */
public void setPosition(int aPosition)
{
    this.position = aPosition;
    this.update("position");
}
```

You have seen that, of all the methods defined for `Frog` objects, only `setPosition()` and `setColour()` set an instance variable directly by using an assignment statement. Such methods are called **setter methods** (or setters). Their sole purpose is to set the value of the appropriate instance variable to the desired value. They may also do things that depend on how the instance variable has been implemented, such as check that the value of the argument is valid.

Setter methods need to access the instance variables directly. Other methods that are used to change the state of a `Frog` object in terms of its `position` or `colour` – such as `left()` or `brown()` – do so indirectly by having the `Frog` object send itself the message `setPosition()` or `setColour()`, by using `this` as the receiver. Notice that the access modifier in the method header for both setter methods is `public`. This is so objects that are not instances of the class `Frog` can send the corresponding messages to alter the instance variables.

Note also that, while subclasses of the `Frog` class inherit the `colour` and `position` instance variables, objects of such a subclass (for example, `HoverFrog`) cannot directly access these instance variables as they have been declared private to instances of the `Frog` class. Hence the setters need to be public so that objects of a subclass can alter their own inherited instance variables. If the setters were to be declared private, then although they would still be inherited by a subclass, they would be hidden from view to that subclass – leaving no way for objects of the subclass to alter their inherited instance variables.

The last statement in the method `setPosition()` is `this.update("position");`. This message-send causes the `update()` method in the superclass (`OUAnimatedObject`) to be executed; this method informs any observing object, such as the graphical display, that the frog has moved position. In M255, the `update()` message is always used within a method whenever it is necessary to trigger an update of a graphical display such as the Amphibians window – this typically happens within a setter method.

We also need methods to get the values of the instance variables: these methods are called **getter methods** (or getters). A getter is a method, such as `getColour()`, whose sole purpose is to return the value of an instance variable. We will look at these methods in Section 3, where we consider messages that return a value as a message answer. Again, the access modifier for these getter methods is `public`, so that the corresponding messages can be sent by objects that are not instances of the `Frog` class and also so that objects that are instances of subclasses of the `Frog` class can access the private inherited instance variables.

In M255, we use the term accessor method to mean either a setter or a getter. Other sources use accessor method to refer to the getter method and use the term mutator method to refer to the setter method.

Setter and getter methods are the two types of **accessor methods**. In other words, they are the methods that enable access to instance variables.

ACTIVITY 9

Suppose that you need to extend the protocol of `Frog` objects to include the message `yellow()`. This new message, when sent to a `Frog` object, will cause the state of the `Frog` object to change so that its instance variable `colour` references `OUColour.YELLOW` (rather than, say, the initial `OUColour.GREEN`). This new message requires a new method, `yellow()`, to be written. In this activity, your task is to write the code for this method. To help you, here is the code for the analogous method `brown()`.

```
/**
 * Sets the colour of the receiver to brown.
 */
public void brown()
{
    this.setColour(OUColour.BROWN);
}
```

Here is the method header for the new method `yellow()`.

```
/**
 * Sets the colour of the receiver to yellow.
 */
public void yellow()
```

Open the BlueJ editor on the `Frog` class in the project Unit4_Project_2. Add your code for the `yellow()` method to the class and recompile. Now test your new method in the OUWorkspace by creating a `Frog` object and sending it the message `yellow()` – view the effect in the graphical display.

DISCUSSION OF ACTIVITY 9

Here is the code that you should have written for `yellow()`.

```
/**
 * Sets the colour of the receiver to yellow.
 */
public void yellow()
{
    this.setColour(OUColour.YELLOW);
}
```

In the OUWorkspace you could use the following statements to test your new method:

```
Frog kermit = new Frog();
kermit.yellow();
```

Exercise 5

Explain what happens to the instance variable `position` when the message `left()` is sent to the object `kermit`.

Solution

The message-send `kermit.left()` causes the code of method `left()`, which is shown below, to be executed.

```
public void left()
{
    this.setPosition(this.getPosition() - 1);
}
```

This entails three steps.

The body of the method `getPosition()` is executed first. The code for the method is given below.

```
public int getPosition()
{
    return this.position;
}
```

Notice the use of `this` to reference the object that was sent the `getPosition()` message, which enables the value of the instance variable `position` stored for that object to be accessed. The value of `position` is then returned as the message answer.

The next step is to take 1 away from the value that has just been returned, and use this value as the argument to `this.setPosition()`.

The final step is to execute the method `setPosition()`. The code for the method is given below.

```
public void setPosition(int aPosition)
{
    this.position = aPosition;
}
```

Again `this` is used, so that the instance variable `position` for `kermit` will be set to the value just calculated in the previous step.

2.4 Access modifiers – public or private?

If an instance variable is declared as private, only instances of the same class can directly access that instance variable. The previous sentence is very important, and you need to appreciate exactly what it means: *if we have two instances of the same class, and one has a reference to the other, then that object can still directly access the other object's instance variable even though that instance variable has been declared as private in the class definition.* In other words, 'private' indicates that an instance variable is only invisible to objects of *other* classes. Instances of other classes cannot access that instance variable directly; even an instance of a subclass that inherits a private instance variable will be unable to access the inherited instance variable directly.

Similarly, if a method is declared as private, only instances of that same class can send the corresponding message to other instances of the class, or indeed to itself; even an instance of a subclass that inherits the method will be unable to send the corresponding message to instances of the superclass, or even to itself.

This can cause problems with inherited instance variables. If some superclass declares an instance variable as private, then if that superclass wishes to allow instances of some subclass to have *any access* to that instance variable, the superclass must make the accessor methods for that instance variable public.

Java has other access modifiers that go some way to ameliorating this problem, as will be briefly examined in *Unit 7*.

2.5 Adding an instance variable to a class

To consolidate what you have learned about **instance variables**, the following three activities will ask you to define an additional instance variable `flyCount` for instances of the class `Frog`, which will be used to record the number of times the message `catchFly()` has been sent to a `Frog` object since the object was created. You will need to initialise `flyCount` and provide **accessor methods**. Also, you will need to add to the code of the method `catchFly()` a statement that will increment the value of `flyCount` every time `catchFly()` is executed.

ACTIVITY 10

Again using the project Unit4_Project_2, open the editor on the `Frog` class.

1 Find the instance variable declarations. Add the variable declaration for `flyCount`, making it private. Remember to include a comment.

2 You now need to initialise `flyCount`. Find the constructor for the `Frog` class and locate the following statements.

```
super();
this.colour = OUColour.GREEN;
this.position = 1;
```

Add a statement to set the initial value of `flyCount` to `0`. Also alter the constructor's comment to reflect what you have done.

3 Recompile the `Frog` class and then create a new `Frog` object in the OUWorkspace and assign it to a variable named `kermit`. Inspect the object referenced by `kermit` by double-clicking on the variable name in the OUWorkspace's list of variables. Check that the new instance variable is in the list of attributes.

DISCUSSION OF ACTIVITY 10

1 Your comment and variable declaration should be something like the following.

```
//Count of the number of flies caught since the Frog was created.
private int flyCount;
```

2 The constructor and its comment should now look like the following.

```
/**
 * Constructor for objects of class Frog, which initialises
 * colour to green, position to 1 and flyCount to 0.
 */
public Frog()
{
    super();
    this.colour = OUColour.GREEN;
    this.position = 1;
    this.flyCount = 0;
}
```

3 To create a `Frog` object you will have executed a statement such as the following.

```
Frog kermit = new Frog();
```

Inspecting the object referenced by `kermit` should show you that `flyCount` has been added to the object's list of attributes, and that `flyCount` has been initialised to `0`.

ACTIVITY 11

Again using Unit4_Project_2, write the accessor methods for `flyCount`. The comments and method headers for the new methods are shown below.

```
/**
 * Returns the flyCount of the receiver.
 */
public int getFlyCount()

/**
 * Sets the flyCount of the receiver to the value
 * of the argument aFlycount.
 */
public void setFlyCount(int aFlyCount)
```

Add the methods `setFlyCount()` and `getFlyCount()` to the `Frog` class. You could use the methods `setPosition()` and `getPosition()` as templates. Note that you do not need a `this.update()` message in the code of your setter method as the value of `flycount` is not represented in the graphical display (the Amphibians window). Test your methods in the OUWorkspace by sending appropriate messages to instances of the `Frog` class.

DISCUSSION OF ACTIVITY 11

You should have something like the following.

```
/**
 * Returns the flyCount of the receiver.
 */
public int getFlyCount()
{
    return this.flyCount;
}

/**
 * Sets the flyCount of the receiver to the value
 * of the argument aFlyCount.
 */
public void setFlyCount(int aFlyCount)
{
    this.flyCount = aFlyCount;
}
```

You could test the new methods with the following code in the OUWorkspace.

```
int count;
Frog kermit = new Frog();
kermit.setFlyCount(3);
count = kermit.getFlyCount();
```

In the OUWorkspace's list of variables you could double-click on either `count` or `kermit` to determine the value of `flyCount`.

The `Frog` class now has a new instance variable, which has been declared and initialised and for which you have written accessor methods. In the next activity you will modify the method `catchfly()` to increment `flyCount`.

ACTIVITY 12

Again using Unit4_Project_2, open the BlueJ editor on the `Frog` class and find the method `catchFly()`. Add a statement to the method that will increment the value of `flyCount` each time the method is executed. Recompile the class and then test your method in the OUWorkspace.

DISCUSSION OF ACTIVITY 12

Here is the additional statement needed in `catchFly()`.

```
this.setFlyCount(this.getFlyCount() + 1);
```

This will increment the instance variable `flyCount` each time the method `catchFly()` is executed.

Note the use of the accessor methods to access the instance variable each time. We would like you to follow this convention in M255. We discuss this further in Section 5.

You could test the modified method with the following statements in the OUWorkspace.

```
Frog kermit = new Frog();
kermit.catchFly();
```

Then you could inspect `kermit` to check the value of the instance variable `flyCount` after each execution of the statement `kermit.catchFly();`.

If you have had difficulties with Activities 6–12, Unit4_Project_3 incorporates all the changes made to the classes in this set of activities.

The `Frog` class now has a new instance variable, `flyCount`, and methods that use this instance variable. We made the accessor methods public. This was necessary so that you could examine, and change, the value of `flyCount` from the OUWorkspace for testing purposes. However, it is worth asking the question: 'should the setter method be public?' In order that the instance variable `flyCount` be meaningful, it should be incremented each time `catchFly()` is executed and not altered at any other time. So the only method that needs access to `setFlyCount()` is `catchFly()`. You could, therefore, make `setFlyCount()` private. If you have time, you might like to try this and check that the method `catchFly()` still works for both `Frog` and `HoverFrog` objects. The getter should remain public, as this does not alter the instance variable `flyCount` and it allows the value of `flyCount` to be used by instances of other classes (and in the OUWorkspace).

In this section you have studied instance variables, and learned how to add them to a class, and seen how to write simple accessor methods. In doing so, you have written a method that requires an argument and a method that returns a value. We now go on to consider such methods in more detail.

3 Methods that return values

In this section we examine how to write methods that return information.

3.1 Message answers and method return values

The effect of sending a message `right()` to an instance of `Frog` is to cause the object to execute the method `right()`, which will change the state of the object so that its `position` instance variable is incremented by 1. This change in state is reflected in the Amphibians window (if you have it open): the `Frog` icon moves one stone to the right. The `right()` message does not return a message answer, but you will recall that there are other messages in the protocol for the `Frog` class that *do* have message answers. An example of such a message is `getColour()`, which, if sent to a `Frog` object, answers with the colour of the receiver, such as `OUColour.GREEN`.

The aim of messages like `getColour()` is to pass back information, not to change the state of the receiver. When such a message is sent to an object, that message then causes a method with the same name to be executed (if it exists in the object's class or any of its superclasses). If the method returns a value, then that value is used as the answer of the message.

So how do you write methods that return values? How do you control what they return? To answer this, look at how `getColour()` is constructed.

```
/**
 * Returns the colour of the receiver.
 */
public OUColour getColour()
{
    return this.colour;
}
```

In the statement

```
return this.colour;
```

the Java keyword `return` instructs the Java system that the value that follows is to be returned by the method. In this case, `colour` is returned, i.e. the value of the `colour` instance variable of the object referenced by `this` (the object that was sent the `getColour()` message). As you learned in Section 2, as with other variables, instance variables can hold references to objects or they can hold the value of a primitive data type. Here, the instance variable `colour` will be a reference to some instance of `OUColour`, for example `OUColour.GREEN`.

You are now going to carry out a series of activities in which you will use and investigate return statements in methods.

ACTIVITY 13

You have just seen that methods that return a value always contain a statement beginning with the Java keyword `return`. Using Unit4_Project_3, use the BlueJ editor to examine the classes `Frog`, `HoverFrog` and `Toad` and make a list of methods that return a value.

DISCUSSION OF ACTIVITY 13

You should have identified the following methods for the classes `Frog` and `Toad`: `getColour()`, `getPosition()`. The class `Frog` also has the method `getflyCount()`. The class `HoverFrog` has a method `getHeight()`. All three classes have a method `toString()`.

Here is the method `getHeight()` for `HoverFrog`.

```
/**
 * Returns the height of the receiver.
 */
public int getHeight()
{
    return this.height;
}
```

Here the method `getHeight()` returns the value of the instance variable `height` of the object referenced by `this` (the object that received the message `getHeight()`). You will recall that the class `HoverFrog` is a subclass of the class `Frog` and inherits all of the instance variables declared in the class `Frog`, but has the instance variable `height` in addition.

Methods that return values provide a way of obtaining a value that depends in some way on the receiver's state. For example, as you saw in Activity 13, this is the case with the method `getHeight()` for `HoverFrog`, where the value returned depends on the value of the instance variable `height`. In the next activity you are going to define a new `Frog` method in which the value returned depends on the value of an instance variable.

The following will be useful in Activity 14: you saw in *Unit 3* that if one of the operands of the + operator is a `String` object then the other operand is automatically converted to a `String`. Thus if a variable `myAge` is defined as type `int` and has the value 21 then `"My age is " + myAge;` evaluates to the `String` `"My age is 21"`.

ACTIVITY 14

Imagine that you wish to obtain the value of the instance variable `position` of a `Frog` object, reported as part of a `String` rather than as an integer. Again using the project Unit4_Project_3, use the BlueJ editor to define a new method `positionReport()` with the following comment and header.

```
/**
 * Return the String "The position is " concatenated with
 * the value of the position of the receiver.
 */
public String positionReport()
```

Once you have recompiled the `Frog` class, satisfy yourself that `Frog` objects respond appropriately to the new message in the OUWorkspace.

DISCUSSION OF ACTIVITY 14

Your method `positionReport()` should look like the following (we've left out the comment).

```
public String positionReport()
{
    return "The position is " + this.getPosition();
}
```

The following statements could be used to test your new method.

```
String frogPosition;
Frog kermit = new Frog();
frogPosition = kermit.positionReport();
```

The string now referenced by the variable `frogPosition` will appear in the Display Pane (if Show Results is checked), or you could check `frogPosition` by inspecting it.

ACTIVITY 15

Again using the project Unit4_Project_3, define and test a similar method named `heightReport()` for instances of the class `HoverFrog`.

DISCUSSION OF ACTIVITY 15

Your method `heightReport()` should look like the following.

```
/**
 * Returns the String "The height is " concatenated with
 * the value of the height of the receiver.
 */
public String heightReport()
{
    return "The height is " + this.getHeight();
}
```

The following statements could be used to test your new method.

```
String frogHeight;
HoverFrog wizzy = new HoverFrog();
wizzy.up();
frogHeight = wizzy.heightReport();
```

Then you could inspect the value of the variable `frogHeight`.

You have now written and tested several methods that explicitly return a value by means of a return statement.

A return statement must be the last statement to be executed in a method. This is a rule that is built into the syntax of the Java language, and the Java compiler will not accept as valid any method that breaks this rule. To illustrate this, you are now going to define a new `Frog` method, which will cause a `Frog` object to move right and then report the new position. Having written this method correctly you will then see what happens if you write the statements in the wrong order.

ACTIVITY 16

Using the project Unit4_Project_3, we would like you to define a new method called `moveRightAndReport()` for the class `Frog`. The code for the method is given below.

```
/**
 * Cause the receiver to move right then return the string
 * "The position is " concatenated with the value of
 * the receiver's position.
 */
public String moveRightAndReport()
{
    this.right();
    return "The position is " + this.getPosition();
}
```

Once you have added this method, recompile the class.

Now rewrite the method with the order of the two statements in the body of the method reversed, i.e.

```
return "The position is " + this.getPosition();
this.right();
```

DISCUSSION OF ACTIVITY 16

When you compile the first version there should be no problems and you should find the method works as expected in the OUWorkspace.

When you rewrite the method with the order of the two statements reversed and click on the Compile button, the compiler should respond by displaying `unreachable statement` in the message area and highlighting the statement `this.right();` in the code. The compiler is indicating that it is a mistake to have a statement after the return statement. We say that the compiler has detected an error in the method that you were attempting to define and cannot compile the new method.

The activities you have just carried out have given you experience in writing return statements to define the value to be returned by a method underlying a message. Subsection 3.2 looks at the method headers for methods that return a value.

3.2 Specifying the return type of a method

As you learnt in *Unit 3*, every variable needs to have a defined **type** so that the compiler knows how much memory to allocate and also so that it can perform some checks for consistency. The return value for a method must also have a defined type. For the method `positionReport()` the return value is of type `String`. This is declared as the second part of the method header. In the next activity you will investigate what happens if the return value is not of the same type as that declared in the method header.

Note that `int`, `float` and `boolean` start with a lower-case letter and `String` starts with an upper-case letter. This is because `String` is a predefined class in the Java language, whereas the other three types are **primitive** data types. At the moment, the main importance this has for you is that you must remember to use upper- or lower-case initial letters as appropriate.

ACTIVITY 17

Still using Unit4_Project_3, open the editor on the `Frog` class and change the return expression for the method `positionReport()` from

```
return ("The position is " + this.getPosition());
```

to

```
return this.getPosition();
```

What happens when you try to compile? Before you leave the editor, correct the method and recompile.

DISCUSSION OF ACTIVITY 17

The Java compiler highlights the return expression and gives the following error message.

```
incompatible types - found int but expected java.lang.String
```

When you correct the method it should compile without an error message.

This is one of the checks that the compiler does when it is compiling your source code into bytecode. It is important to realise that, although the compiler highlights the return statement here, it could have been the type declaration in the method header that was incorrect. When you get this error message you need to check both places and decide which type you need the return value to be.

If you have had difficulties with Activities 13–17, Unit4_Project_4 incorporates all the changes made to the classes in this set of activities.

SAQ 6

What is the type of the return value for the method `getColour()`?

ANSWER.....

It is of type `OUColour`.

SAQ 7

Write down the method header for a method that has the name `myMethod`, is accessible to any object, has no arguments, and returns an integer value.

ANSWER.....

```
public int myMethod()
```

3.3 Messages with no reply

SAQ 8

Think back over the methods you have looked at in the class `Frog`. Which methods do not have a return value? Look at the method headers for these methods. What do they all have in common?

ANSWER

The methods `setPosition()`, `sameColourAs()`, `setColour()`, `brown()`, `green()`, `yellow()`, `croak()`, `home()`, `left()`, `right()`, `jump()`, `catchFly()`, `setFlyCount()` do not have a return value. They all have the keyword `void` as the second part of their header.

The keyword `void` is used to tell the Java compiler not to expect a return value from the method. If a return statement appears in a method with the keyword `void` then the compiler will generate an error message. Conversely, if a method has a return type that is *not* `void`, then the method must include a return statement, and the value returned must be consistent with the type declared in the method header.

SAQ 9

Write down the method header for a method that has the name `myMethod`, is accessible to any object, has no arguments, and has no return value.

ANSWER

```
public void myMethod()
```

Exercise 6

Using pen and paper, write a new method called `distanceFromHome()` for the `Frog` class, which will return the value of the position of the receiver minus 1 (1 being the home position).

Solution

```
/**
 * Return the position of the receiver minus 1 (the home position).
 */
public int distanceFromHome()
{
    return this.getPosition() - 1;
}
```

4 Methods with arguments

In Activity 8 you discovered that setter methods use an assignment statement to change the value of the appropriate instance variable. To do this they need to have the new value of the instance variable given to them. This is done by means of an **argument**.

In this section we investigate how to write methods for messages that require arguments. If a message needs an argument then the corresponding method will declare an identifier to handle the argument provided by the message. This declaration gives the type of the argument that the message should provide. Some messages have more than one argument, and therefore their corresponding methods have more than one such declaration.

The activities in this section use the project Unit4_Project_4.

4.1 Formal arguments

You will recall that, in the Amphibian Worlds of *Unit 2*, some messages require information to be supplied when sending a message. For example, when using the message `sameColourAs()` you must specify which instance of `Frog`, `HoverFrog` or `Toad` the message should take as an example of the desired colour. So you might have written (and executed) the statement

```
frog1.sameColourAs(frog2);
```

in order to get `frog1` to change its state so that it is the same colour as `frog2`. We say that `frog2` is the argument of the message. And if you want `frog1` to change its colour to that of (say) `frog3`, then you send a similar message but with `frog3` as the argument:

```
frog1.sameColourAs(frog3);
```

SAQ 10

Write down the receivers and the arguments of the following message-sends.

(a) `frog1.sameColourAs(frog3);`

(b) `frog4.setPosition(3);`

ANSWER

(a) `frog1` is the receiver and `frog3` is the argument.

(b) `frog4` is the receiver and `3` is the argument.

Here is the method `setPosition()`.

```
/**
 * Sets the position of the receiver to the value of the argument aPosition.
 */
public void setPosition(int aPosition)
{
    this.position = aPosition;
    this.update("position");
}
```

The method header here is

```
public void setPosition(int aPosition)
```

The name of the method is `setPosition()` and the declaration `int aPosition` inside the parentheses indicates that this method expects an argument. The argument `aPosition` given in the method header is called a **formal argument**. In this case, the formal argument has the name (identifier) `aPosition` and it is declared as being of the primitive type `int`. Note that the declaration of a formal argument is similar to the declaration of a variable, and you can think of the formal argument as being a special sort of variable. Note that the type of a formal argument can also be the name of some class.

When the message `setPosition()` is sent to a `Frog` object an actual argument is used. This actual argument must be type compatible with the formal argument in the method header of the corresponding method. So in the message

```
frog4.setPosition(3);
```

the integer `3` is the actual argument for the message.

On receiving the message `setPosition(3)`, the `Frog` object referenced by `frog4` causes the method `setPosition()` to be executed with its formal argument `aPosition` set to the value of message's actual argument, in this case `3`.

ACTIVITY 18

You have just learned that methods that take arguments have their arguments declared in the method header. Using the BlueJ editor and Unit4_Project_4, make a list of the methods that take arguments in the `Frog` class.

DISCUSSION OF ACTIVITY 18

You should have identified the following methods: `setPosition()`, `sameColourAs()`, `setColour()` and `setFlyCount()`.

When sending a message, the type (or class) of the actual argument of a message must be type compatible with the type of the formal argument of the corresponding method.

Exercise 7

What do you think will happen when you execute the following statements in the OUWorkspace?

```
Frog frog1 = new Frog();
frog1.setPosition(OUColour.RED);
```

Solution

`OUColour.RED` is not an integer, so an error message will be displayed in the Display Pane of the OUWorkspace. In fact, the actual error message would be:

```
Semantic error: Message setPosition(ou.OUColour) not understood
by class 'Frog'.
```

The name of the method together with the parentheses and the type of any arguments is called the **method signature**. The signature of `setPosition()` is `setPosition(int)`. The signature shows clearly the type of the expected argument.

Note that the `Frog` class defines two methods with the name `sameColourAs()`. One has the formal argument `aFrog` of class `Frog` and the other has the formal argument `aToad` of class `Toad`. If there were a single `sameColourAs()` method whose formal argument was declared as being of class `Frog`, then we would only be able to send a `sameColourAs()` message to a `Frog` object with an actual argument that was an instance of `Frog` (or a subclass of `Frog`). We would not be able to send a message `sameColourAs()` to a `Frog` object with an actual argument that was an instance of `Toad` – hence the necessity for the second method.

Thus the two methods `sameColourAs()` have the signatures `sameColourAs(Frog)` and `sameColourAs(Toad)`.

Note that two methods within the same class cannot have the same signature. For example, two hypothetical methods with the following method headers:

```
public void doSomething(int marks)
private int doSomething(int commission)
```

have the same signature, namely `doSomething(int)`, hence the Java compiler would report an error if you attempted to compile a class that defined two such methods.

ACTIVITY 19

With Unit4_Project_4 open, select Save As from the BlueJ Project menu and save the project with a different name, such as Unit4_Project_4_Act_19. You will need to recompile after doing this, so press the Compile button on the project window. After you have done this, open the OUWorkspace and from its Graphical Display menu open the Amphibians window.

Execute the following statements one at a time in the OUWorkspace.

```
Frog kermit = new Frog();
Frog gribbit = new Frog();
Toad toad1 = new Toad();
kermit.setColour(OUColour.RED);
gribbit.sameColourAs(kermit);
gribbit.sameColourAs(toad1);
```

Now execute the following two statements in the OUWorkspace.

```
HoverFrog wizzy = new HoverFrog();
gribbit.sameColourAs(wizzy);
```

DISCUSSION OF ACTIVITY 19

The first set of statements should all work correctly, and you should end up with `gribbit` being brown.

The second set of statements (using a `HoverFrog` object as the actual argument) should also work and `gribbit` should return to green.

ACTIVITY 20

Still using the project you saved in Activity 19, use the BlueJ editor to delete the method `sameColourAs()`, which has a formal argument type of `Toad`, from the class `Frog`.

Recompile the class `Frog` and execute in the OUWorkspace the statements given in Activity 19 again. What happens this time?

Close the project when you have finished this activity.

DISCUSSION OF ACTIVITY 20

You should have found that the message

```
gribbit.sameColourAs(toad1);
```

no longer works and you get the error message

```
Semantic error: Message sameColourAs(Toad) not understood by class
'Frog'
```

However, the message

```
gribbit.sameColourAs(wizzy);
```

should still work correctly.

Look at the code below for the version of the method `sameColourAs()` with a `Toad` object as its formal argument.

```
/**
 * Sets the colour of the receiver to the argument's colour.
 */
public void sameColourAs(Toad aToad)
{
    this.setColour(aToad.getColour());
}
```

In the method `sameColourAs()` above, `aToad` is the formal argument. When the method is executed (as the result of a `sameColourAs()` message), the formal argument `aToad` will reference a `Toad` object provided as the actual argument of the message `sameColourAs()`.

Exercise 8

Suppose you wish to add a new message `increasePosition()` to the protocol of instances of the `Frog` class. The effect of this message will be to increase the value of the attribute `position` by an amount defined by an argument, which we shall call `step`.

Using pen and paper, write a suitable comment and method header for a method called `increasePosition()` that will support this behaviour.

Now (again using pen and paper) write the code for the method body. Remember to use the accessor messages for the instance variable. You can use the method `right()` as a guide.

Solution

Your method should look something like the following.

```
/**
 * Increases the value of the position of the receiver by
 * the value of the argument step.
 */
public void increasePosition(int step)
{
    this.setPosition(this.getPosition() + step);
}
```

SAQ 11

What is the signature of the above method?

ANSWER.....

The signature is `increasePosition(int)`.

ACTIVITY 21

Reopen Unit4_Project_4, and open the editor on the `Frog` class.

Add your new method `increasePosition()` and compile.

Satisfy yourself that `Frog` and `HoverFrog` objects respond appropriately by executing message-sends in the OUWorkspace, such as `frog1.increasePosition(3)`. Remember that you can check visually on the behaviour of `Frog` and `HoverFrog` objects if you open the graphical display.

DISCUSSION OF ACTIVITY 21

The single statement in the method body of `increasePosition()` requires some explanation. It includes the two messages `getPosition()` and `setPosition()` as well as the arithmetic operator `+`.

Consider what happens if the following message is sent to the object referenced by `frog1` when its instance variable `position` has the value `3`.

```
frog1.increasePosition(5);
```

- On receiving the message `increasePosition()`, the object referenced by `frog1` causes the method `increasePosition()` to execute. The method's formal argument, `step`, is set to the value of the message's actual argument, which is `5`.
- The message `getPosition()` produces an answer that is the value of the current `position` of the receiver, `frog1`, i.e. the integer `3`.
- The integer `3` is then added to the actual argument, `5`, by means of the arithmetic operator `+`, and results in the integer `8`.
- The integer `8` becomes the actual argument for the `setPosition()` message sent to the receiver `this`, so the instance variable `position` of the object referenced by `frog1` is set to `8`.

Exercise 9

Describe what happens when the following message is sent to the `Frog` object referenced by `frog2`, whose instance variable `position` has the value `6`.

```
frog2.increasePosition(4);
```

Solution.....

- On receiving the message `increasePosition()`, the object referenced by `frog2` causes the method `increasePosition()` to execute. The method's formal argument, `step`, is set to the value of the message's actual argument, which is `4`.
- The message `getPosition()` produces an answer that is the value of the current `position` of the receiver, `frog2`, i.e. the integer `6`.

- The integer `6` is then added to the actual argument, `4`, by means of the arithmetic operator `+`, and results in the integer `10`.
- The integer `10` becomes the actual argument for the `setPosition()` message sent to the receiver `this`, so the instance variable `position` of the object referenced by `frog2` is set to `10`.

4.2 Methods with two arguments

Activities 22 and 23 introduce a method with two arguments. In a method with more than one formal argument, the arguments are separated by commas.

ACTIVITY 22

Suppose that there is (another) requirement for `Frog` objects – to set both the position of a frog to a given value and its colour to a given instance of `OUColour` with a single message. To implement this behaviour you will need a new method, `setPositionAndColour()`, which will take two arguments. One argument will give the position the frog must go to and the other argument will give the colour that the frog must assume.

Using the project Unit4_Project_4, write the code for this new method for the `Frog` class using the BlueJ editor, then recompile.

Then, using the OUWorkspace, satisfy yourself that `Frog` and `HoverFrog` objects respond appropriately to the new message you have added to the protocol of the `Frog` class.

DISCUSSION OF ACTIVITY 22

Here is our version of `setPositionAndColour()`.

```
/**
 * Sets the position of the receiver to the argument aPosition
 * and the colour of the receiver to the argument aColour.
 */
public void setPositionAndColour(int aPosition, OUColour aColour)
{
    this.setPosition(aPosition);
    this.setColour(aColour);
}
```

You could have tested your code in the OUWorkspace with statements such as the following.

```
frog1.setPositionAndColour(5, OUColour.BLUE);
frog3.setPositionAndColour(1, OUColour.MAGENTA);
```

SAQ 12

What is the signature of the method given in the discussion of Activity 22?

ANSWER.....

The signature is `setPositionAndColour(int, OUColour)`.

ACTIVITY 23

In this activity you look at what happens if the order of the arguments in a message is not the same as the order defined in the method header of the corresponding method.

With the project Unit4_Project_4 and the OUWorkspace open, execute the following statements one by one. What happens?

```
Frog kermit = new Frog();
kermit.setPositionAndColour(3, OUColour.BLUE);
kermit.setPositionAndColour(OUColour.BLUE, 3);
```

DISCUSSION OF ACTIVITY 23

The first two statements execute normally and (if you have the graphical display open) you will see the `Frog` object, `kermit`, change position and colour.

The third statement does not execute. Instead, an error message is displayed indicating that the compiler cannot find a corresponding method with the signature `setPositionAndColour(OUColour, int)`.

You can see from this that the order in which the arguments are defined in the method header is important. The actual arguments of the corresponding message must be in the same order.

4.3 A more significant example

We conclude this section with an activity that gives you practice with the topics covered in the section.

ACTIVITY 24

Using the project Unit4_Project_4, you are required to alter the `Frog` class so that the value for the home position can be stored using another instance variable of type `int` called `homePosition`. This instance variable will be initialised to `1` (so `Frog` objects will still be created in the same initial state). However, it will be possible to change the home of a frog to another position.

In each of the following steps make sure you provide adequate documentation in the form of comments.

1 Add the instance variable declaration for `homePosition` in the correct place in the class code.

2 Add the code to initialise `homePosition` to `1` in the constructor.

3 Add the accessor methods `setHomePosition()` and `getHomePosition()` at the end of the class, and then compile so that your class is ready for testing.

4 Test your work so far by executing the following statements one by one in the OUWorkplace and inspecting the temporary variable, `hPos`.

```
int hPos;
hPos = 0;
Frog frog1 = new Frog();
hPos = frog1.getHomePosition(); //check hPos is now 1
frog1.setHomePosition(5);
hPos = frog1.getHomePosition(); //check hPos is now 5
```

5 Alter the method `home()` so that it sets the value of `position` to the value of `homePosition`. You will need to recompile after this alteration.

6 Test your work by executing the following statements one by one in the OUWorkspace and inspecting the variable `pos`.

```
int pos;
pos = 0;
Frog frog1 = new Frog();
frog1.setHomePosition(3);
frog1.home();
pos = frog1.getPosition(); //check pos is now 3
```

7 Regenerate the documentation for Unit4_Project_4 by selecting Project Documentation from the BlueJ Tools menu.

DISCUSSION OF ACTIVITY 24

1 The instance variable declarations should now look like the following.

```
private OUColour colour;
private int position;
private int flyCount;
private int homePosition;
```

2 The code for the constructor should now look like the following.

```
/**
 * Constructor for objects of class Frog which initialises
 * colour to green, position to 1, flyCount to 0 and
 * homePosition to 1.
 */
public Frog()
{
    super();
    this.colour = OUColour.GREEN;
    this.position = 1;
    this.flyCount = 0;
    this.homePosition = 1;
}
```

3 The accessor methods should be similar to the following.

```
/**
 * Returns the home position of the receiver.
 */
public int getHomePosition()
{
    return homePosition;
}

/**
 * Sets the home position of the receiver to the value of
 * the argument aPosition.
 */
public void setHomePosition(int aPosition)
{
    this.homePosition = aPosition;
}
```

4 You should find that a new `Frog` object has an initial `homePosition` of `1`, and that you can use the accessor methods to get and set the value of `homePosition`.

5 The method `home()` should now look like the following.

```
/**
 * Resets the receiver to its home position.
 */
public void home()
{
    this.setPosition(this.getHomePosition());
}
```

6 After executing the code you should find that `pos` holds the value `3`.

7 You should find that the documentation for the `Frog` class now includes information on the new methods.

5 Reuse of code

So far in this unit we have introduced techniques that enable Java programmers to write code in an object-oriented manner. In this section we introduce a major concept in object-oriented programming – reuse of code.

You have seen that the code for the `Frog` class includes several methods. Each method can be tested to ensure that it conforms to the specification given in the method comment. When you add a new method to a class you can reduce some of the complexity of the new method, and also make the new method easier to maintain, by reusing methods that have already been developed and tested for that class.

Consider the method `doubleLeft()` for the `Toad` class, which you met in Activity 5:

```
/**
 * Moves the receiver left twice.
 */
public void doubleLeft()
{
    this.left();
    this.left();
}
```

Now look at some alternative code for this method.

```
/**
 * Moves the receiver left twice.
 */
public void doubleLeft()
{
    this.setPosition(this.getPosition() - 4);
}
```

You might think this is better. However, consider the requirement stated in the method comment.

```
/**
 * Moves the receiver left twice.
 */
```

This, and the name of the method, `doubleLeft()`, tell us that the method is supposed to have the same effect as a double application of `left()`.

At present, the `Toad` class defines `left()` to mean changing the position by 2, and so `doubleLeft()` changes the position by 4. So for the time being the statement

```
this.setPosition(this.getPosition() - 4);
```

achieves the correct result.

But suppose the designers of the `Toad` class changed their minds, as they easily might, and decreed that toads moved left 3 stones at a time! 'Move left twice' would now mean changing position by 6, and the statement `this.setPosition(this.getPosition() - 4);` would be wrong and would need to be altered. If the designers had a further change of mind, another modification would be needed, and so on.

However, the earlier version of `doubleLeft()`, which uses `this.left()` twice, would continue to work perfectly without any changes! It will always produce the same effect as two moves left. The designers of `Toad` could change their minds about what `left()` means as many times as they like, and this version of the method would always automatically produce the correct result, without requiring any modification.

This is very important, because there could be many different methods whose result depends on the definition of 'move left'. If the methods are written so that they manipulate the position directly, then every time there is a change to the method `left()` you would need to identify all those methods and update their code. Quite apart from the effort involved, you might overlook some and it would be easy to make mistakes.

However, methods that *reuse* `left()` will not require any changes at all, and all these problems are entirely avoided! Reuse is a far superior strategy, and experienced object-oriented programmers employ reuse wherever they can. Although you are not yet an expert, you should look carefully for possible reuse and try to take advantage of it in the code you write.

5.1 Using libraries

The ability to reuse code goes beyond the simple example described above.

One of the features of object-oriented languages that make them so powerful is that they usually include class libraries. Such libraries may contain hundreds, or even thousands, of classes that have proven useful to programmers in the past to write a wide range of applications. These classes have been robustly tested, and they have been well documented, so that programmers can understand the purpose of each class and what each method in a class does, without having to know how the methods are implemented. Indeed, class libraries usually do not include source code, just the compiled code, so the programmer has to rely on the documentation as they cannot see the source code.

Java comes with many such class libraries (which it calls packages); in addition, programmers can develop their own class libraries (packages) containing classes that are relevant to their own application area. Once tested and documented, these can be reused to form the basis of many applications.

You are already making use of a library of classes (package) developed for M255. It is this reuse of code that enables you to have frogs moving in a graphical display at this early stage in the course. In the next activity, we look at the documentation for the simplest of these classes. As this is an exploratory activity, there is no discussion.

ACTIVITY 25

Open the BlueJ editor on the `Frog` class in the project Unit4_Project_4. At the top of the editor window the first statement is:

```
import ou.*;
```

This tells the compiler to find the library (package) called ou and make all the classes in it available to the `Frog` class. Now select OU Class Library from the BlueJ Help menu. The documentation for the library will soon appear in a browser window. You will notice that the package name is `ou` and that it contains four classes. `OUAnimatedObject` is the superclass of the classes `Frog` and `Toad`, and it contains the methods that enable `Frog`, `HoverFrog` and `Toad` objects to be visible and move around a graphical display. `OUColour` is the class that provides the colours we have used in our classes. `OUDialog` provides methods that you will use in *Unit 5* to communicate with a user via dialogue boxes. The fourth class, `OUFileChooser`, is not used until Unit 12.

Click on the link for `OUColour` in the left-hand frame. We have been using this class each time we write a statement such as

```
freddie.setColour(OUColour.GREEN);
```

Do not worry if you do not understand the detail. All you need to appreciate is that the various colours we use are kept in this class as constants. You will learn more about this in *Unit 7*. There is only one instance method in this class: `toString()`.

Click on the link for `OUAnimatedObject` in the left-hand frame of your browser window. This is a much more complicated class, but it still contains many parts you might recognise, including a list of methods that include `update()` and `performAction()`. Now, going back to the BlueJ editor, look at the code for the `Frog` class. You will see that we use `update()` in the methods `setPosition()` and `setColour()`, and we use `performAction()` in the methods `jump()` and `croak()`. There is no need to worry about how these methods are implemented. We have all the information we need in the documentation provided for the library – a description of what they do and the method header. Someone else has put in the time to code these methods and test them to ensure they work.

Later in the course we will make extensive use of the Java Class Libraries. You can see the documentation for these libraries in your browser by selecting Java Class Libraries from the Help menu in the BlueJ window. You do not need to access this documentation now, but you might like at some stage to look at what is provided. Depending on how your computer is set up you may need to be connected to the internet in order to access this documentation.

5.2 Using accessor methods

Earlier in this section you saw that you could rewrite the method `doubleLeft()` without reusing `left()`, although this might not be advisable. It is even possible to rewrite the method without using the accessor methods at all, since we can work with the variable `position` directly, although you will see shortly that there are strong reasons why this is even less advisable. The resulting method would be as follows.

```
/**
 * Moves the receiver left twice.
 */
public void doubleLeft()
{
    this.position = this.position - 4;
    this.update("position");
}
```

This seems fine – noting the use of `update()` to notify any observing user interface, such as the Amphibians window, that the receiver's position has changed. However, what happens if you write a new method `tripleLeft()` and forget the statement containing the message `update()`? You will investigate this in the next activity.

ACTIVITY 26

Using the project Unit4_Project_4, open the BlueJ editor on the `Toad` class and add the following method.

```
/**
 * Moves the receiver left three times.
 */
public void tripleLeft()
{
    this.position = this.position - 6;
}
```

Recompile, open the OUWorkspace from the BlueJ Tools menu, and then open an Amphibians window from the Graphical Display menu.

Execute the following statements one at a time in the OUWorkspace.

```
Toad dermot = new Toad();
dermot.tripleLeft();
```

Inspect the state of `dermot` by double-clicking on the name `dermot` in the Variables Pane in the OUWorkspace. Is the state of `dermot` reflected in the Amphibians window?

DISCUSSION OF ACTIVITY 26

You should have found that the value of `position` for `dermot` had changed to `5` but that the icon of the toad representing `dermot` in the Amphibians window was still sitting on the 11th stone.

If you have had difficulties with Activities 18–26, Unit4_Project_4_Completed incorporates all the changes and additions made in this set of activities.

Advantages of using accessor methods

When developing the OUWorkspace for this course, we arranged for the Amphibians window to display a representation of any amphibian-like objects created in the OUWorkspace (i.e. the domain model of the Amphibians window is the collection of amphibian objects in the OUWorkspace). However, the Amphibians window and the collection of objects in the OUWorkspace have to be kept in step. Therefore any method that directly changes an instance variable that is of importance to the Amphibians window must also send a message to notify that window (or indeed any other interested observing object) that the value of an object's instance variable has changed. If this is not done, the Amphibians window will not know to refresh itself to reflect any changes to the state of that object. This is exactly what has happened with the method you tested in Activity 26.

You will have noticed that the setter methods `setColour()`, `setPosition()` and `setHeight()` all contain statements to notify an observing object (in this case the Amphibians window) that a particular object's instance variable has changed its value. This has the important consequence that a method that makes use of an instance variable's setter, rather than changing the instance variable directly, will have the Amphibian window notified automatically that the instance variable has changed.

The need to notify some observing object that the value of some other object's instance variable has changed is not the only reason why it is often better to use accessor methods than to work with instance variables directly. What would happen if, for some reason, the class were changed so that `position` was stored as a `float`

rather than an `int`? The programmer making the changes would ensure that the method headers of the accessors remained identical, so that the getter would still answer with an integer, and the setter would still take an integer argument. The change in number format would be strictly internal to the methods, and any number conversions that were required would be taken care of. Methods that depended on the accessors would go on working precisely as before.

However, methods that used the instance variables directly might not be in this fortunate situation. They might rely on `position` being an integer, and the change of number format might 'break' them so that they stopped working. If programmers had used direct access instead of accessor methods, it would be necessary to track down every place where this had happened, check how the method was affected and rewrite the code if necessary. This obviously entails a lot of additional work, and is an error-prone process.

For reasons that cannot be elaborated upon here, it is safer, where possible, to initialise instance variables directly in a constructor rather than using setter messages.

So when should a programmer reuse accessor methods, and when is it good practice to access instance variables directly? This is not an easy question. Direct access is simpler, and can take less processing time than using accessor methods. However, setters may have checks for validity of the arguments, and both setters and getters may mask the fact that the actual implementation of *how* an instance variable is stored may have changed. In M255, you are not going to design any time-critical software so, as a rule of thumb, we suggest that wherever accessors are available you should use them in your methods. This will mean the only place – other than in a constructor – where an instance variable is assigned directly is in the setter method, and the only place it is accessed directly is in the getter method. These two methods – the getter and setter – *have* to operate directly with the instance variable, because there is no other way for them to work. They cannot use accessors, because they *are* the accessors!

To recap, there are two major advantages in using accessor methods (if they are provided).

1 Using accessor methods is more economical in terms of coding. This is because you can define the detailed work required in changing some aspect of an object's state just once with a setter method, and then ensure that you make use of that detailed work in other methods by reusing the setter.

2 Accessor methods aid reuse. That is, they facilitate the business of changing a class or creating a subclass for use in a different situation. For example, the implementor of a subclass does not need to know the details involved in updating or returning the values of inherited instance variables. Indeed, an object of some subclass is prevented from directly accessing an inherited instance variable if that instance variable has been declared as private, and has to make use of accessor methods that the superclass must declare as public (so that instances of the subclass can see them). This means that the subclass is insulated from any changes that might be made in the superclass to the representation of its instance variables, or to the detailed work required in changing some aspect of its state. Changes may be made to the superclass without the knowledge of any subclasses, provided the specifications of the getter and setter methods are preserved. So in the case of the class `Frog`, an instance of a subclass – or indeed an object of any class – can access the instance variable `position` using `getPosition()` and be confident that the message will always answer with an `int` value.

   It is for reasons such as these – minimising the impact of change – that it is sensible, even in simple applications, to use accessor messages (getters and setters for attributes) instead of accessing instance variables directly.

6 Encapsulation

So far in this unit you have seen how an object contains both state (the values of its attributes held in instance variables) and behaviour (defined by its message protocol). This is an example of what is termed **encapsulation**. Put another way, objects *encapsulate* state and behaviour. The concept of encapsulation is very powerful because it allows an efficient division of labour in large software projects. Each team member can work in isolation on the class(es) he or she is responsible for. All that team members need to know about other classes are the names and specifications of the public methods.

Related to encapsulation is the concept of **data hiding**, whereby an object becomes a black box, with access to the encapsulated data (the instance variables) being possible *only* through a limited set of public methods. In other words, only an object's own methods are allowed to access the value of an instance variable (to either change it or return it).

Objects in Java exhibit encapsulation and they can also exhibit data hiding. You have seen how it is possible to define access modifiers for each instance variable and method. If the instance variables for an object are defined as `private`, then that object can exhibit data hiding as well as encapsulation. However, other instances of the same class *will* be able to directly access the private instance variables so, to some extent, Java relies on the individual programmer to enforce data hiding.

Figure 6 illustrates the concept of encapsulation and data hiding.

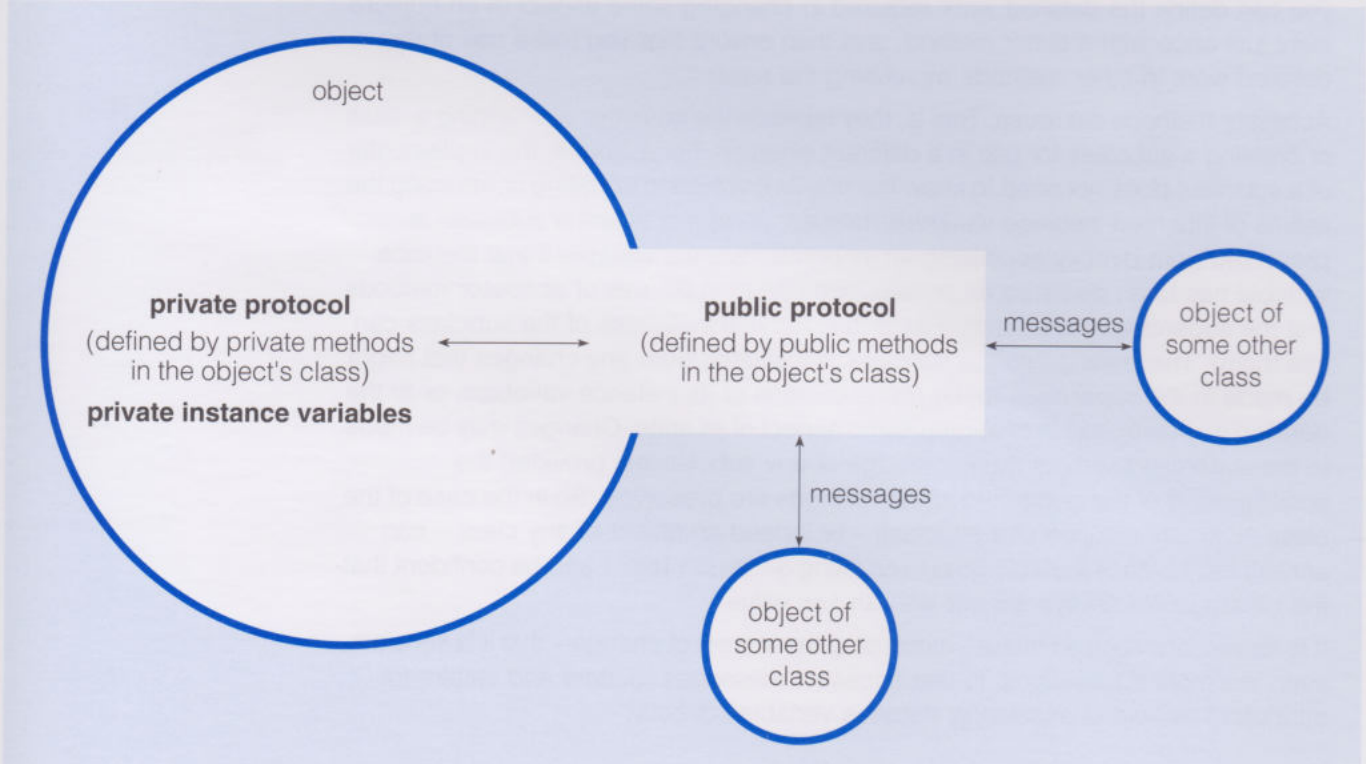


Figure 6 Encapsulation

Do you remember the millennium bug – Y2K? Some operating systems and some programming languages record time and dates as just another number. (Compare this with Java, which has a `Date` class provided in the `java.util` package.) As time progresses the time number gets bigger – so a future date is always larger than a past date. The Y2K bug arose because many programmers writing commercial systems in the 1970s and 1980s had omitted the century digits from dates – such programmers did not think the century digits would be needed! For example, many COBOL programs represent the date 26 February 1990 as the number 900226, and the date 1 January 1991 as 910101, allowing the programs to compare the two numbers and correctly assume that the smaller number represented the earlier date. However, without the century digits, those comparisons would fall apart on 1 January 2000. The last day of the millennium was 991231, and after the stroke of midnight many computers would see 1 January 2000 as 000101 – a smaller number than the day before. Time would appear to have been reversed! To avoid this disaster a lot of money was spent paying people to read programs line-by-line, checking how the programs implemented dates and, if need be, modifying the programs to include century digits. The cost of all this work is estimated to have exceeded 400 billion pounds worldwide! If all software had been written in an object-oriented language (or at least a language providing encapsulation and data hiding) the cost would have been considerably less, as it would only have been necessary to check objects that implemented dates.

SAQ 13

How is encapsulation implemented in Java?

ANSWER.....

In Java you can define classes that are templates for the creation of objects. These objects then encapsulate both state and behaviour.

SAQ 14

How do you implement data hiding in Java?

ANSWER.....

Data hiding is implemented by declaring instance variables as `private`.

7 Consolidation

This section contains activities designed to help you practise the techniques and explore some of the concepts you have learned in this unit. The activities are all based on the project Unit4_Project_5. Unless you leave your computer during your study of this section, we suggest that you do not close the project or the OUWorkspace during this set of activities. The project Unit4_Project_5 contains the classes `Circle`, `Triangle` and `Diamond`. It also contains a partially complete class called `Marionette`.

7.1 The `Marionette` class

First, you need to familiarise yourself with the `Marionette` class.

ACTIVITY 27

Launch BlueJ and open the project Unit4_Project_5. From the BlueJ Tools menu select Project Documentation (if a dialogue box opens, click on its Regenerate button). When the browser opens to display the generated Javadoc documentation, select the link for the `Marionette` class in the left-hand frame. Now look at the documentation for the class in the main frame of the browser window. What methods can you see?

DISCUSSION OF ACTIVITY 27

The visible methods are `addComponents()` and `right()`.

ACTIVITY 28

With the project Unit4_Project_5 open, choose Open from the Graphical Display menu of the OUWorkspace. This will open up a graphics window called Shapes, in which shape-like objects created in the OUWorkspace can be displayed.

Execute the following statements in the OUWorkspace one at a time and check they do what you might expect, both by inspecting the variables that reference the objects you create and by looking at their graphical representations in the Shapes window.

```
Circle head = new Circle();
Diamond body = new Diamond();
Triangle leftLeg = new Triangle();
Triangle rightLeg = new Triangle();
Marionette mary = new Marionette();
mary.addComponents(head, body, leftLeg, rightLeg);
mary.right(20);
mary.right(20);
```

DISCUSSION OF ACTIVITY 28

On creating each shape (a circle, a diamond and two triangles) you should have seen that they were all displayed in the top-left of the Shapes window, all on top of each other. As soon as the `addComponents()` message was sent to the `Marionette` object referenced by `mary` you should have seen that the shapes became arranged into a recognisable marionette-type figure. Finally, sending `right()` messages to `mary` caused this marionette figure to move to the right.

The class `Marionette` is incomplete. Objects of this class have no arms and can move in only one direction (right). In the next two activities you will look at the code for the class `Marionette` and add the necessary instance variables and methods to complete the required behaviour for its instances.

ACTIVITY 29

1 With the project Unit4_Project_5 open, open the BlueJ editor on the class `Marionette`. Notice that there are many more methods in the code than appeared in the documentation. Why do you think this is? Why do they not appear in the documentation? What would happen if you tried to send a message that corresponded to one of these methods to an instance of `Marionette` from the OUWorkspace?

2 Look at the code for the method `right()` and write similar methods `left()`, `up()` and `down()`. Test these in the OUWorkspace. Why should these methods be public?

DISCUSSION OF ACTIVITY 29

1 The methods that have been declared as private do not appear in the documentation. If you tried to send these messages from the OUWorkspace you would get an error message. The methods have been made private because the corresponding messages are not to be part of the public behaviour of a marionette; rather, they exist to 'help' the publicly visible methods. You will notice that all the `Marionette` instance variables and their accessor methods have been declared as private. This has an important consequence: only `Marionette` objects can access the instance variables, either directly or indirectly. This means also that any instance of some future subclass of `Marionette` would be unable to access these instance variables, either directly or indirectly.

2 Here is our code for the methods `left()`, `up()` and `down()`.

```
/**
 * Decrements xPos by the value of the argument.
 */
public void left(int decrement)
{
    this.setXPos(this.getXPos() - decrement);
}
```

Although it seems initially counter-intuitive, the `up()` method needs to decrement `yPos` (and conversely `down()` needs to increment `yPos`) because, in Java, the origin (0,0) is at the top left of the working area.

```
/**
 * Decrements yPos by the value of the argument.
 */
public void up(int decrement)
{
    this.setYPos(this.getYPos() - decrement);
}

/**
 * Increments yPos by the value of the argument.
 */
public void down(int increment)
{
    this.setYPos(this.getYPos() + increment);
}
```

You could test your methods by executing the following statements one by one in the OUWorkspace.

```
Circle head = new Circle();
Diamond body = new Diamond();
Triangle leftLeg = new Triangle();
Triangle rightLeg = new Triangle();
Marionette mary = new Marionette();
mary.addComponents(head, body, leftLeg, rightLeg);
mary.left(50);
mary.up(50);
mary.down(50);
```

These methods need to be public to allow other objects (including you, from the OUWorkspace) to send the corresponding messages as they are part of the new public behaviour for `Marionette` objects.

ACTIVITY 30

The body parts of a `Marionette` object are referenced by the following instance variables: `body`, `head`, `rightLeg` and `leftLeg`.

1 You are now going to add arms to a `Marionette` object. The arms of `Marionette` objects should be instances of class `Diamond`. Look at the code for declaring the instance variable `leftLeg`. Write similar declarations for two new instance variables: `leftArm` and `rightArm`.

2 Next, using `setLeftLeg()` and `setRightLeg()` as templates, write the setter methods `setLeftArm()` and `setRightArm()` for these new instance variables. You should set the heights of both arms to `15`, their widths to `40`, and their colour to `OUColour.YELLOW`. Once you have done this, write the getter methods for the new instance variables.

3 In the method `addComponents()` the instance variables that form the body parts are assigned the objects given in the actual arguments of a corresponding message.

Add formal arguments to the argument list in the method header of `addComponents()` for the new arms. Then add code to the `addComponents()` method to set the new `rightArm` and `leftArm` instance variables to these additional arguments.

4 In order to align the arms of a `Marionette` object to the rest of its body parts you will need to write two more methods: `alignLeftArm()` and `alignRightArm()`. Use the methods `alignLeftLeg()` and `alignRightLeg()` as templates. For the left arm you should to set its `xPos` to the `xPos` of the marionette minus 35, and its `yPos`

to the `yPos` of the marionette plus 25. For the right arm you will need to set its `xPos` to the `xPos` of the marionette plus 25, and its `yPos` to the `yPos` of the marionette plus 25.

5 Finally, you will need to add two more statements to the `alignAll()` method that make use of `alignLeftArm()` and `alignRightArm()`. Once you have done this, create a `Marionette` object in the OUWorkspace and send it `right()`, `left()`, `up()` and `down()` messages to make sure that the code you have written in this activity works correctly.

DISCUSSION OF ACTIVITY 30

1 You should have added the following code to the list of instance variables.

```
private Diamond leftArm;
private Diamond rightArm;
```

2 The code for these instance variables' accessor methods should be as follows.

```
/**
 * Sets the leftArm of the receiver to the argument and
 * then sets the height, width and colour of leftArm.
 */
private void setLeftArm(Diamond anArm)
{
    this.leftArm = anArm;
    this.leftArm.setHeight(15);
    this.leftArm.setWidth(40);
    this.leftArm.setColour(OUColour.YELLOW);
}

/**
 * Returns the leftArm of the receiver.
 */
private Diamond getLeftArm()
{
    return this.leftArm;
}

/**
 * Sets the rightArm of the receiver to the argument and
 * then sets the height, width and colour of rightArm.
 */
private void setRightArm(Diamond anArm)
{
    this.rightArm = anArm;
    this.rightArm.setHeight(15);
    this.rightArm.setWidth(40);
    this.rightArm.setColour(OUColour.YELLOW);
}

/**
 * Returns the rightArm of the receiver.
 */
private Diamond getRightArm()
{
    return this.rightArm;
}
```

3 Here is our updated code for the method `addComponents()`.

```
/**
 * Sets the head, body, leftLeg, rightLeg, leftArm and rightArm of the
 * receiver to the arguments. Then causes these instance variables to be
 * aligned relative to each other.
 */
public void addComponents(Circle aHead, Diamond aBody, Triangle
aLeftLeg, Triangle aRightLeg, Diamond aLeftArm, Diamond aRightArm)
{
    this.setHead(aHead);
    this.setBody(aBody);
    this.setLeftLeg(aLeftLeg);
    this.setRightLeg(aRightLeg);
    this.setLeftArm(aLeftArm);
    this.setRightArm(aRightArm);
    this.alignAll();
}
```

4 Here is our code for the methods `alignLeftArm()` and `alignRightArm()`.

```
/**
 * Aligns the leftArm of the receiver relative to the xPos and yPos of the
 * receiver.
 */
private void alignLeftArm()
{
    this.leftArm.setXPos(this.getXPos() - 35);
    this.leftArm.setYPos(this.getYPos() + 25);
}

/**
 * Aligns the rightArm of the receiver relative to the xPos and yPos of the
 * receiver.
 */
private void alignRightArm()
{
    this.rightArm.setXPos(this.getXPos() + 25);
    this.rightArm.setYPos(this.getYPos() + 25);
}
```

5 Below is the updated code for the method `alignAll()`.

```
/**
 * Aligns all the body parts of the receiver.
 */
private void alignAll()
{
    this.alignBody();
    this.alignHead();
    this.alignLeftLeg();
    this.alignRightLeg();
    this.alignLeftArm();
    this.alignRightArm();
}
```

Finally, here are statements you could have used in the OUWorkspace to test that the new code worked correctly.

```
Circle head = new Circle();
Diamond body = new Diamond();
Triangle leftLeg = new Triangle();
Triangle rightLeg = new Triangle();
Diamond leftArm = new Diamond();
Diamond rightArm = new Diamond();
Marionette mary = new Marionette();
mary.addComponents(head, body, leftLeg, rightLeg, leftArm, rightArm);
mary.left(50);
mary.up(50);
mary.down(50);
```

If you have had difficulties with Activities 27–30, Unit4_Project_5_Completed incorporates all the changes and additions made in this set of activities.

8 Summary

After studying this unit you should understand the following ideas.

- A method defines a message for all instances of a class and for all instances of any subclasses.
- Although a method has to be defined only once to work for a whole class of objects, when a message causes a method to be executed, the method behaves as though it is inside exactly one object – namely the object that received the message that caused the execution of the method.
- Attributes are implemented in Java as instance variables. An instance variable holds either a primitive data value or a value that is a reference to an object. The declared type of the variable specifies what kind of value it can hold.
- Access modifiers are used to specify the visibility of instance variables and methods to objects of other classes.
- You can *think* of two kinds of things existing inside an object: its own instance variables and its own methods. All other objects (and their methods) can be considered to be outside a given object. This incorporation of data and behaviour in a single entity is termed encapsulation.
- The only way that an object (or a user) can change the state of another object or make it do something is by sending it a message. If you have followed good practice and made all the instance variables private, there is no direct access to the state of an object from objects of other classes. It is up to programmers to respect encapsulation between objects of the same class.
- Objects or primitive data values are returned as values from a method by writing a return statement in the method code.
- Arguments are used in a message to pass information to an object's method. The formal argument names in the heading of a method are placeholders for the actual arguments.

LEARNING OUTCOMES

After studying this unit you should be able to:

- explain the distinction between a method and a message;
- explain what is meant by the principles of encapsulation and reuse of code;
- use an editor to create new methods for a class, and test the methods on a variety of instances of that class;
- add instance variables to a class, initialise them and write accessor methods;
- write a simple constructor for a class;
- write methods that return values;
- understand the use of arguments in messages, and write code for methods involving arguments;
- explain the meaning of public and private in relation to both instance variables and methods.

Glossary

accessor method A method that implements an **accessor message**. See **getter method** and **setter method**.

accessor message An accessor message is either a **getter** or a **setter** message. For example, the messages `getPosition()` and `setPosition()` are accessor messages for the **instance variable** `position` held by instances of the `Frog` class. The getter message `getPosition()` returns the value of the instance variable `position`, while the setter message `setPosition()` changes the value of `position`.

argument Some messages require information. For example, when requesting a `Frog` object to change its colour to that of another `Frog` object, it is necessary to specify the other `Frog` object. This is seen in the message-send `frog1.sameColourAs(frog2)`. A message can have zero, one or more arguments. The arguments (if any) used in a message are known as *actual arguments*, since they contain actual values or references to actual objects that will be passed to the method via the corresponding **formal arguments**. The actual arguments must match the formal arguments in number and type, and must appear in the same order.

bytecode Bytecode is the intermediate code produced by the Java **compiler**. In BlueJ, compilation is done when the Compile button is pressed. This will create a bytecode file, for example `Frog.class`, from the source code file `Frog.java`. The bytecode file is portable, because each computer that can run Java programs has a Java Virtual Machine – a program itself – that understands bytecode and converts it into the machine code required for that particular computer.

comment A comment is a piece of text in program code that is ignored when executing the code. In Java multi-line comments are delimited by `/*` and `*/`. Single line comments are simply preceded by `//`. A comment can generally be placed anywhere in the code of a class, with the exception of method comments – method comments are placed between `/**` and `*/` and must appear *immediately before* the method header.

compiler A piece of software which first checks that text written in a high-level language is correctly formed. If the check is successful, then the source code is compiled into **bytecode**.

data hiding This is where an object is treated as a black box, with access to the encapsulated data (the **instance variables**) being possible *only* through a limited set of methods, i.e. only an object's own methods are allowed to access the value of an instance variable (either to change it or return it).

debugging The identification and removal of implementation errors (bugs) from a program.

encapsulation Objects allow you to *encapsulate* data by incorporating into a single entity (the object) both the data (**instance variables**) and the behaviour (methods) defined for that data. The concept of encapsulation is very powerful because it allows an efficient division of labour in large software projects. Each team member can work in isolation on one or more classes. The only things that team members need to know about other classes are the names and specifications of the methods.

formal argument An identifier used in a method to stand for a value that is passed into the method by a message.

getter method An **accessor method** whose purpose is to return the value of an **instance variable** as its message answer.

instance variable A variable that is common to all the instances of a class but the value of which is specific to each instance. Each instance variable either contains a reference to an object or contains a primitive data value. For example, the instance variable `colour` contains a reference to an instance of the class `OUColour`, say `OUColour.GREEN`; whereas the `position` instance variable contains a value of type `1`.

Javadoc A program that comes with Java. The Javadoc program picks up information from specially formatted comments and other parts of the class code such as the constructor and the **method headers**. These are all used to create an HTML file, which describes the class in a standard way. This description is aimed not at the Java compiler, but at human readers (and possibly the writer of the code at a later date, when he or she might well have forgotten what the methods do).

method The code that is invoked by the Java Virtual Machine at run-time when an object receives a message.

method body That part of a **method** enclosed by braces that follows the **method header**.

method header A method header consists of an access modifier (e.g. `public`), a return value (e.g. `int` or `void`) and a name (e.g. `setPosition`) followed by the formal argument names enclosed in parentheses (e.g. `(int aNumber)`). For example, the method header for a method whose name is `setPosition()` is `public void setPosition(int aNumber)`.

method invocation At run-time, selecting and executing a **method** when an object receives a message.

method signature The name of the **method** together with the parentheses and the types of any arguments. For example, the signature for the `setPosition()` method in the `Frog` class is `setPosition(int)`.

`private` An access modifier. It tells the Java compiler that the only objects that have access are the object to which it belongs and other objects of the same class.

pseudo-variable A special undeclared variable, visible within a method or constructor, that cannot be changed by assignment. Java has two such variables – `this` and `super`.

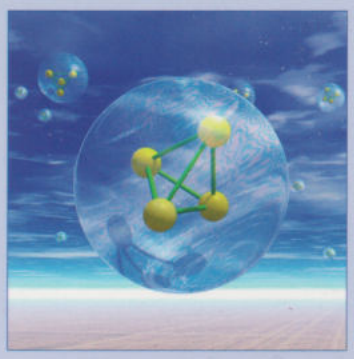
`public` An access modifier. It tells the Java **compiler** that all objects have access.

setter method An **accessor method** whose purpose is to assign a new value to an instance variable. The new value is determined by the single argument of the method.

signature See **method signature**.

`this` A **pseudo-variable** used within a **method** to reference the receiver of the message that activated the method.

Index



M255 Unit 4
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

UNIT 4

Block 1

M255 Unit 4
ISBN 978 0 7492 5496 4